中山大学 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7讲：语法分析(3)

张献伟

xianweiz.github.io

DCS290, 3/21/2024

中山大学 SUN YAT-SEN UNIVERSITY

Review Questions

- Grammar G: $stmt \rightarrow$ **if** ($expr$) $stmt$ **else** $stmt$
  | **while** ($expr$) $stmt$ | v
  $expr \rightarrow$ 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 a non-terminal symbol.
- Is **while** (false) **else** *v* an sentence of G?

  NO. It cannot be derived using the production rules.
- Grammar G: $E \rightarrow T \,/\, E \mid T$, result of 6 - 4 / 2?
  $T \rightarrow T - T \mid$ id

  (6 - 4) / 2 = 1
- Regard id - id - id, is G ambiguous?

  Yes. No associativity is specified for operator -.

Example

a, b = 1, c = 2;
a = b = c;
b = a - b - c;
b = ???

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  int a, b = 1, c = 2;
  a = b = c;

  b = a - b - c;

  printf("\t\t\ta=%d, b=%d\n", a, b);

  a = 0, b = 2, c = 3;
  printf("\t\t\t(0==2-3) = %d, (0==2!=3) = %d\n", a == b - c, a == b != c);

  return 0;
}
```

```
# vim ari.c
# clang -o ari ari.c
# ./ari
 a=2, b=-2
```

```
(0==2-3) = 0, (0==2!=3) = 1
```

+ -	Binary plus(Addition) Binary minus(subtraction)	Left to right
== !=	Equal to Not equal to	Left to right

https://www.programiz.com/c-programming/precedence-associativity-operators

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

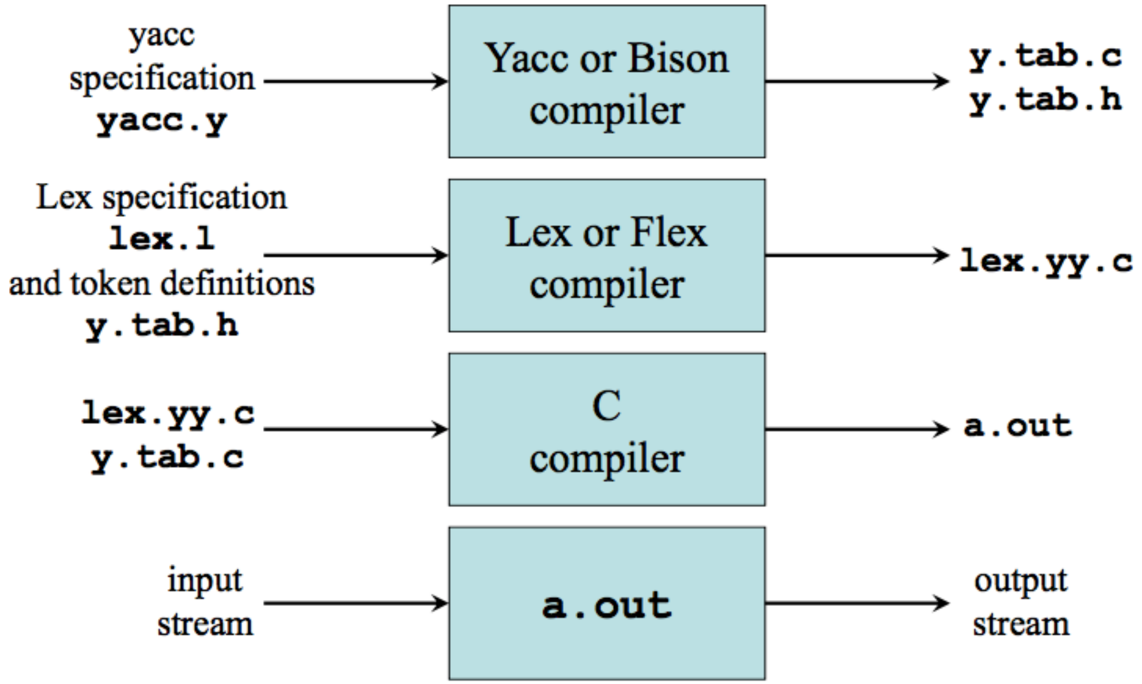
```
%{
 #define YYSTYPE double
 #include "y.tab.h"
 extern double yylval;
%}
number [0-9]+\.?|[0-9]*\.[0-9]+

%%

[ ]          { /* 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

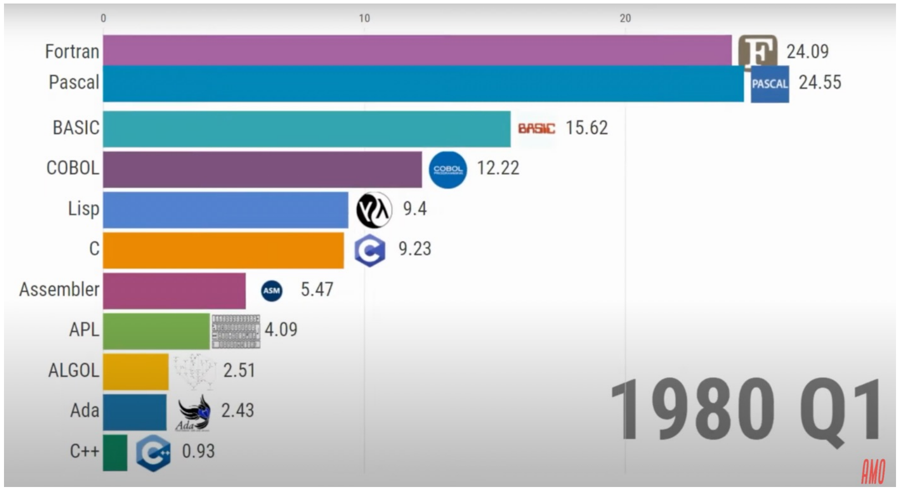
- Run
  - $./*test* < exprs.txt

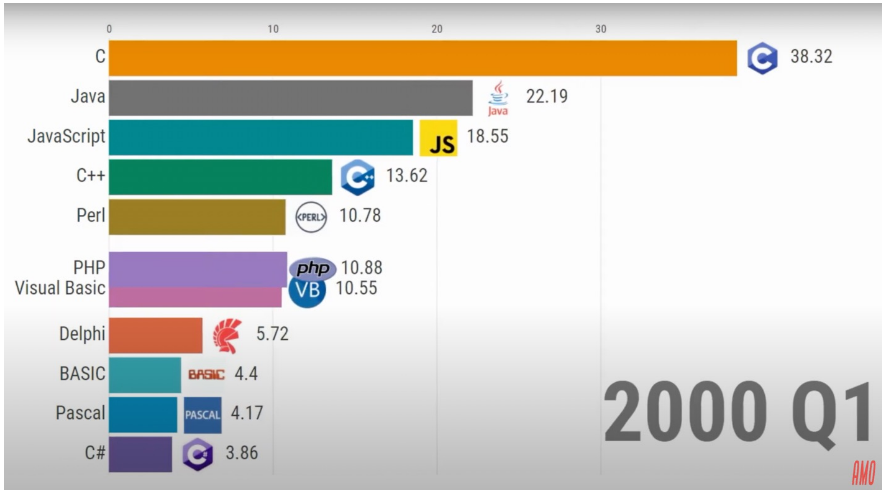
```
1 + 5
1 * 2 + 10
10 - 2 -3
```

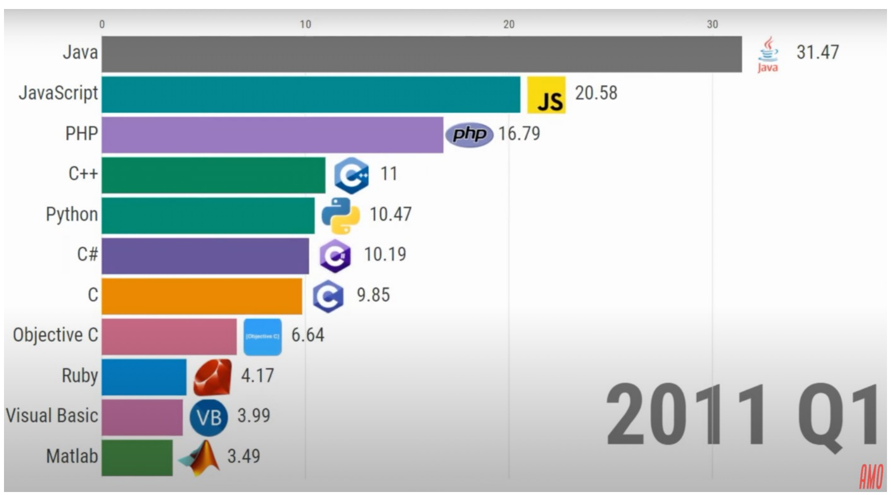
Detour

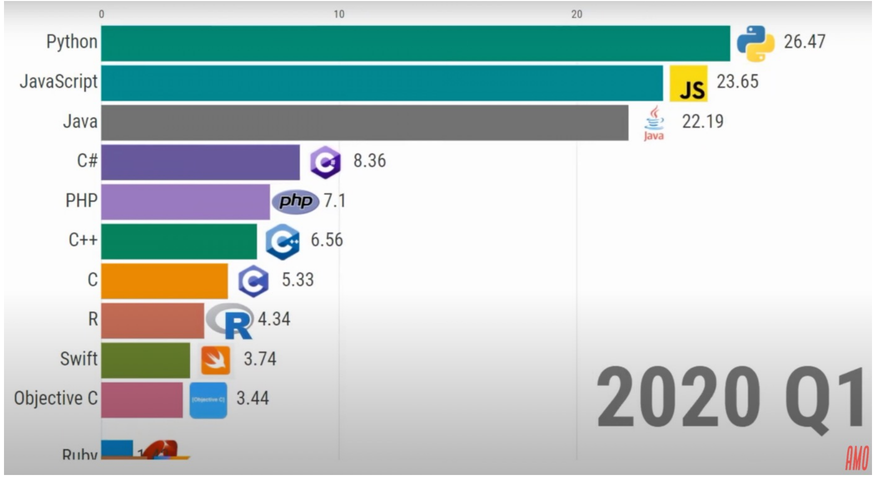
- **Lexer**
  - Lex: initial release in 1975
  - Flex (fast lexical analyzer generator): written around 1987
- **Parser**
  - Yacc: full description was published in 1975
  - GNU Bison: initial release in 1985
  - ANTLR (ANother Tool for Language Recognition): initial release in 1992
- **Compiler**
  - GCC (GNU Compiler Collection): initial release in 1987
  - LLVM/Clang: initial release in 2003/2007
- **Language**
  - C/C++/Python/Java/Rust: first appeared in 1972/1985/1991/1995/2015

Detour (cont.)



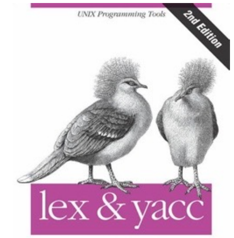




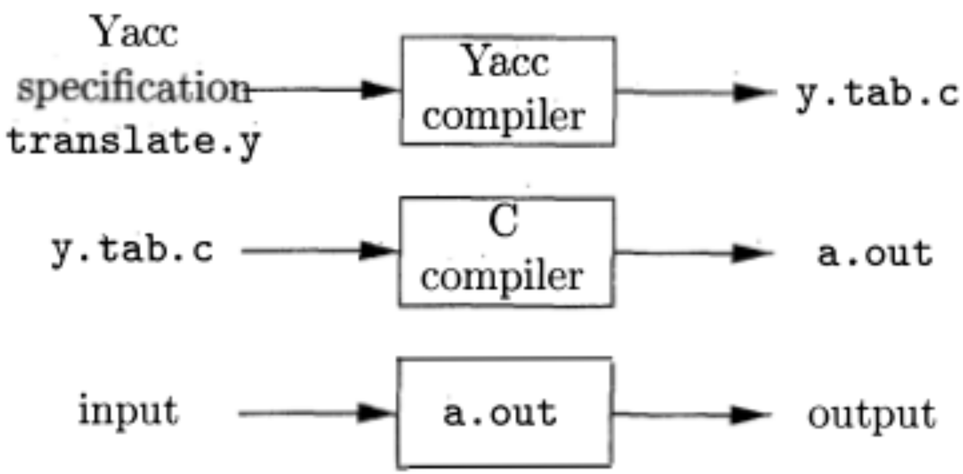


https://www.youtube.com/watch?v=t04YDL1oFkU

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 $\langle head \rangle \rightarrow \langle body \rangle_1 \mid \ldots \mid \langle body \rangle_n$ would be written in YACC as:

  $$\begin{array}{l} \langle head \rangle : \langle body \rangle_1 \ \{ \langle semantic\ action \rangle_1 \} \\ \qquad\qquad \ldots \\ \qquad : \langle body \rangle_n \ \{ \langle semantic\ action \rangle_n \} \\ \qquad ; \end{array}$$

- 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*

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

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

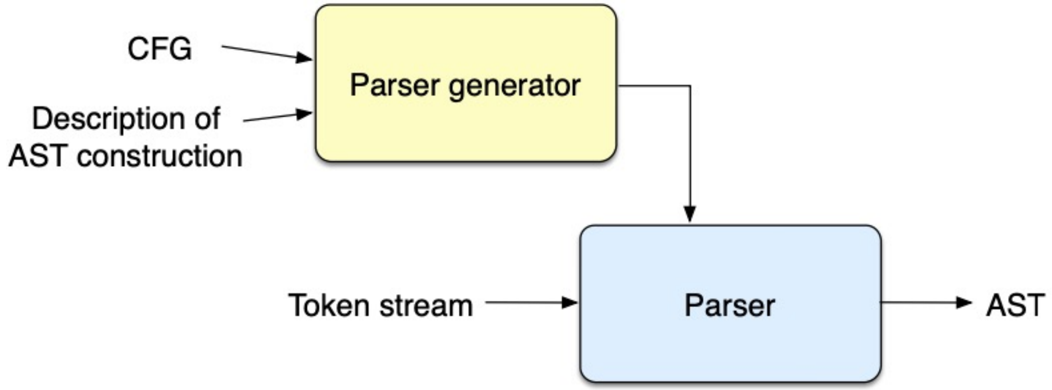
```
%{
 #include <ctype.h>
 #include <stdio.h>
 #define YYSTYPE double /* double type for Yacc stack */
%}
%token NUMBER

%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
     ;
```

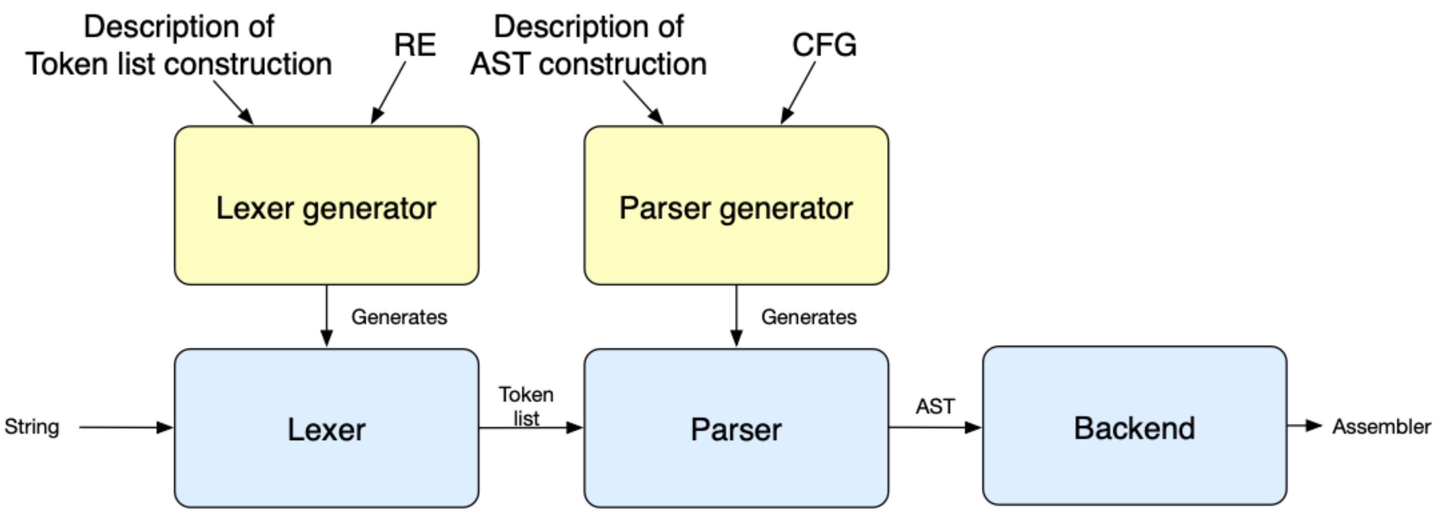
Can we remove those two lines?

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

ε

Parser

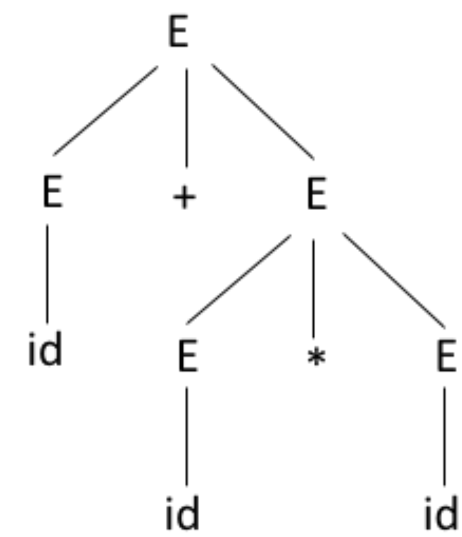




https://users.sussex.ac.uk/~mfb21/compilers/slides/5.pdf

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?[哪个符号？]
    - Which production of the non-terminal to use?[哪个规则？]
  - 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→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)

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)

S

SUN YAT-SEN UNIVERSITY

Example

- Consider a CFG grammar G

  S→AB　　A→aC　　B→bD　　D→d　　C→c

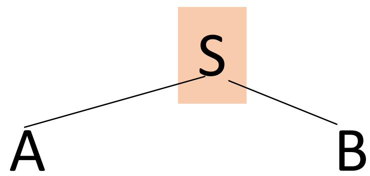
- This language has only one sentence: L(G) = {acbd}

Top-down (leftmost derivation)

S ⇒ 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)



Example

- Consider a CFG grammar G

  S→AB   A→aC   B→bD   D→d   C→c

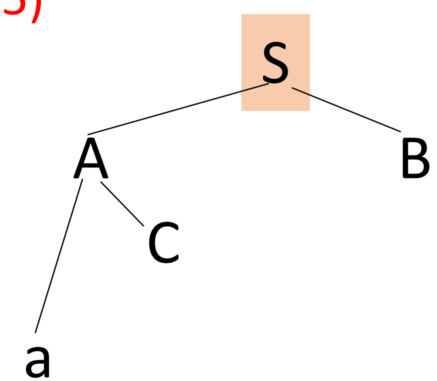
- This language has only one sentence: L(G) = {acbd}

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)



Example

- Consider a CFG grammar G

  S→AB   A→aC   B→bD   D→d   C→c

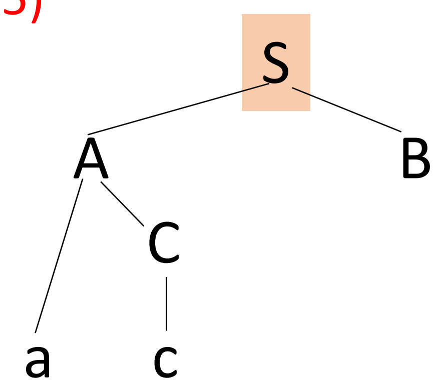
- This language has only one sentence: L(G) = {acbd}

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)



Example

- Consider a CFG grammar G

  S→AB   A→aC   B→bD   D→d   C→c

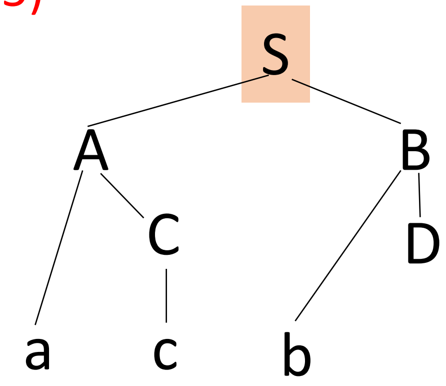
- This language has only one sentence: L(G) = {acbd}

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)



Example

- Consider a CFG grammar G

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

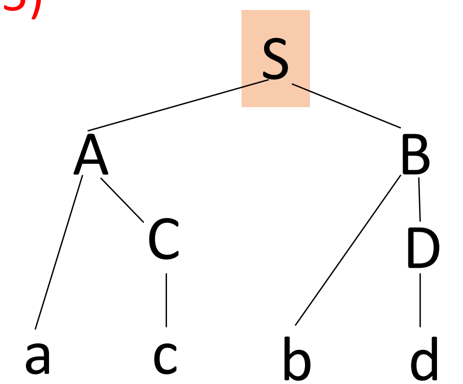
- This language has only one sentence: L(G) = {acbd}

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 ⇒ AB (1)

⇒ aCB (2)

⇒ acB (3)

⇒ acbD (4)

⇒ acbd (5)

Bottom-up (reverse of rightmost derivation)

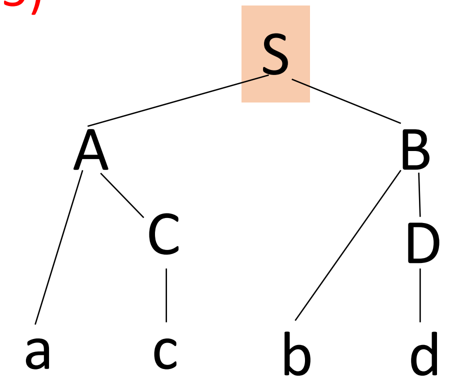
S ⇒ AB (5)

⇒ AbD (4)

⇒ Abd (3)

⇒ aCbd (2)

⇒ acbd (1)



a c b d

Example

- Consider a CFG grammar G

  S→AB A→aC B→bD D→d C→c

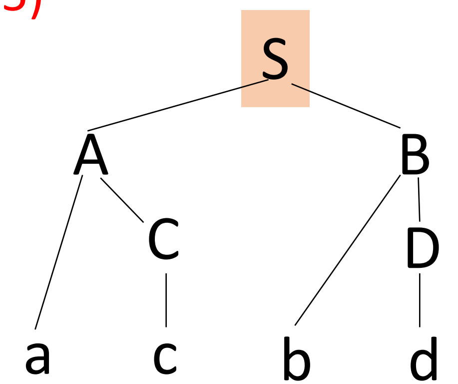
- This language has only one sentence: L(G) = {acbd}

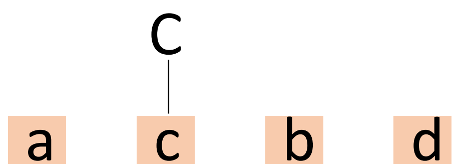
Top-down (leftmost derivation)

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)





Example

- Consider a CFG grammar G

  S→AB A→aC B→bD D→d C→c

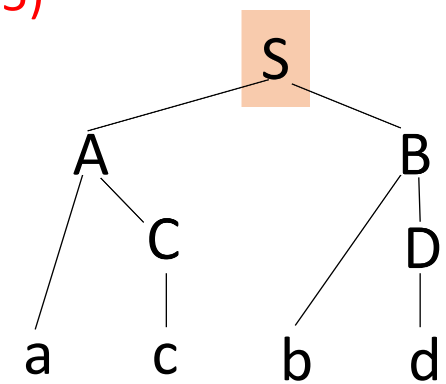
- This language has only one sentence: L(G) = {acbd}

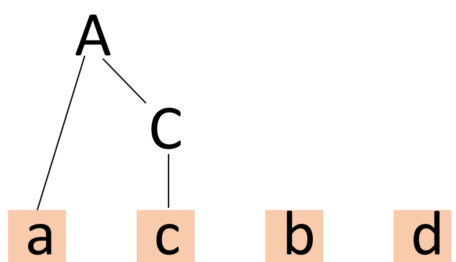
Top-down (leftmost derivation)

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)





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)

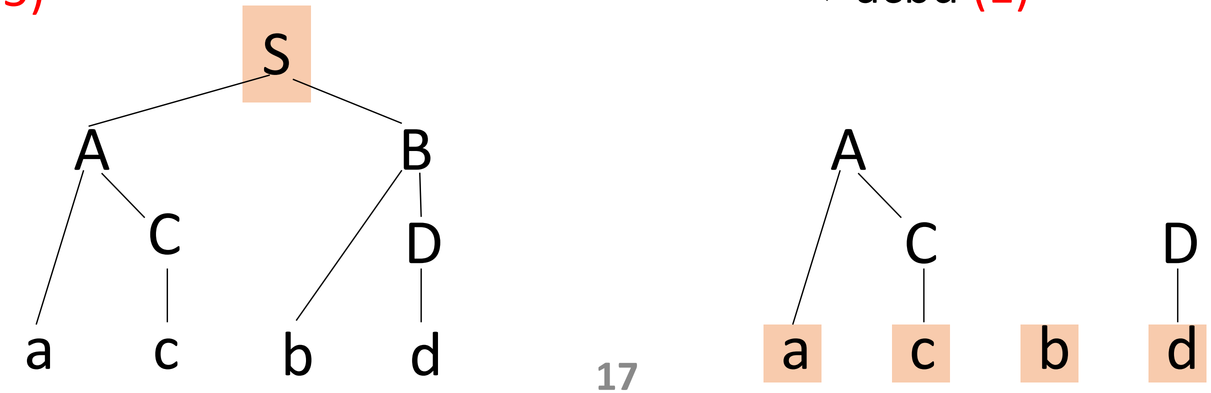
$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

$\Rightarrow aCbd$ (2)

$\Rightarrow acbd$ (1)



Example

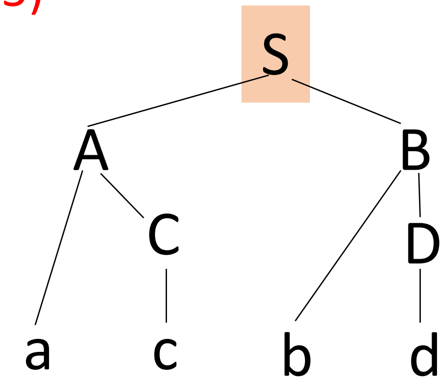
- Consider a CFG grammar G

  S→AB   A→aC   B→bD   D→d   C→c

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

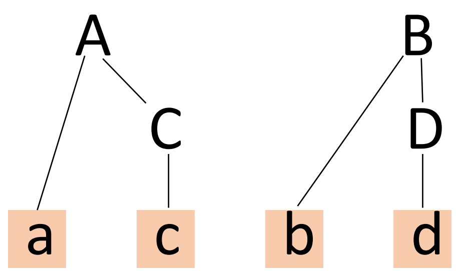
Top-down (leftmost derivation)

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)



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)

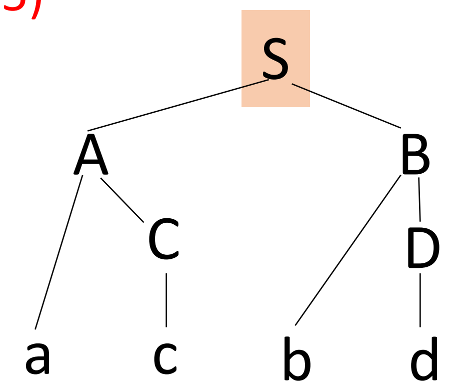
S ⇒ AB (1)

⇒ aCB (2)

⇒ acB (3)

⇒ acbD (4)

⇒ acbd (5)



Bottom-up (reverse of rightmost derivation)

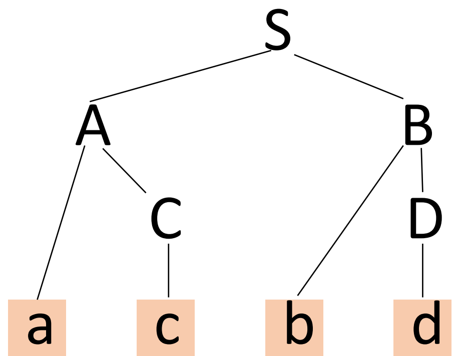
S ⇒ AB (5)

⇒ AbD (4)

⇒ Abd (3)

⇒ aCbd (2)

⇒ acbd (1)



Preview: Bottom-up Parsing[自底向上]

- 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)

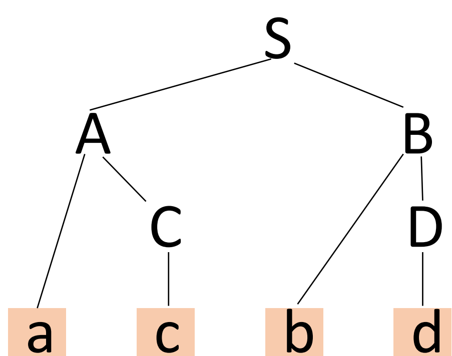
S ⇒ AB (5)

⇒ AbD (4)

⇒ Abd (3)

⇒ aCbd (2)

⇒ acbd (1)



Top-down Parsers[自顶向下]

- **Recursive descent parser** (RDP, 递归下降分析) with backtracking[回溯]
  - Implemented using recursive calls to functions that implement the **expansion** of each non-terminal[非终结符-展开]
  - Goes through all possible expansions by **trial-and-error** until match with input; backtracks when mismatch detected[试错-回溯]
  - Simple to implement, but may take exponential time
- **Predictive parser**[预测分析]
  - Recursive descent parser with prediction (no backtracking)
  - Predict next rule by looking ahead *k* number of symbols
  - Restrictions on the grammar to avoid backtracking
    - Only works for a class of grammars called LL(k)

Classify rule: for a non-terminal, which production to use?

RDP with Backtracking[回溯]

- **Approach**: for a non-terminal in the derivation, productions are tried in some order until[N: 展开]
  - A production is found that generates a portion of the input, or[向前推进]
  - No production is found that generates a portion of the input, in which case backtrack to previous non-terminal[向后回溯]

- Terminals of the derivation are compared against input[T: 比较]
  - Match: advance input, continue parsing[能对上，向前推]
  - Mismatch: backtrack, or fail[对不上，向后退]

- Parsing fails if no derivation generates the entire input

Example

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - Begin with a tree consisting of a single node labeled *S*

Example

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - Begin with a tree consisting of a single node labeled *S*

S

Example

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - Begin with a tree consisting of a single node labeled *S*
  - The input pointer pointing to *c*, the first symbol of *w*

S

Example

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - Begin with a tree consisting of a single node labeled *S*
  - The input pointer pointing to *c*, the first symbol of *w*

S

Example

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - Begin with a tree consisting of a single node labeled *S*
  - The input pointer pointing to *c*, the first symbol of *w*
  - *S* has only one production, so we use it to expand *S* and obtain the tree

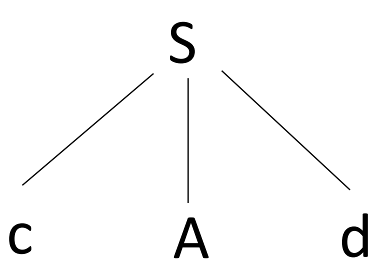
S

Example

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - Begin with a tree consisting of a single node labeled *S*
  - The input pointer pointing to *c*, the first symbol of *w*
  - *S* has only one production, so we use it to expand *S* and obtain the tree

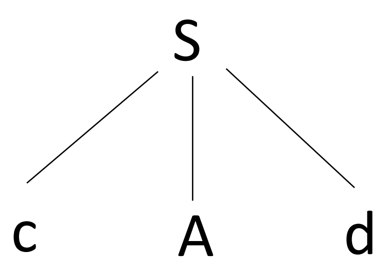


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - … …
  - The leftmost leaf, labeled *c*, matches the first symbol of *w*
    - So we advance the input pointer to *a* (i.e., the 2nd symbol of *w*) and consider the next leaf *A*

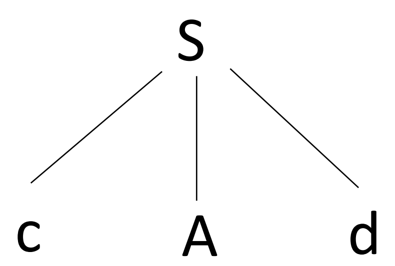


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - … …
  - The leftmost leaf, labeled *c*, matches the first symbol of *w*
    - So we advance the input pointer to *a* (i.e., the 2nd symbol of *w*) and consider the next leaf *A*

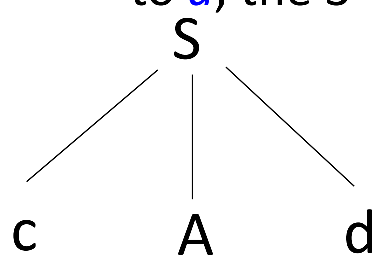


Example (cont.)

- Consider the grammar

  S → cAd A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - … …
  - The leftmost leaf, labeled *c*, matches the first symbol of *w*
    - So we advance the input pointer to *a* (i.e., the 2nd symbol of *w*) and consider the next leaf *A*
  - Next, expand *A* using *A* → *ab*
    - Have a match for the 2nd input symbol, *a*, so advance the input pointer to *d*, the 3rd input symbol

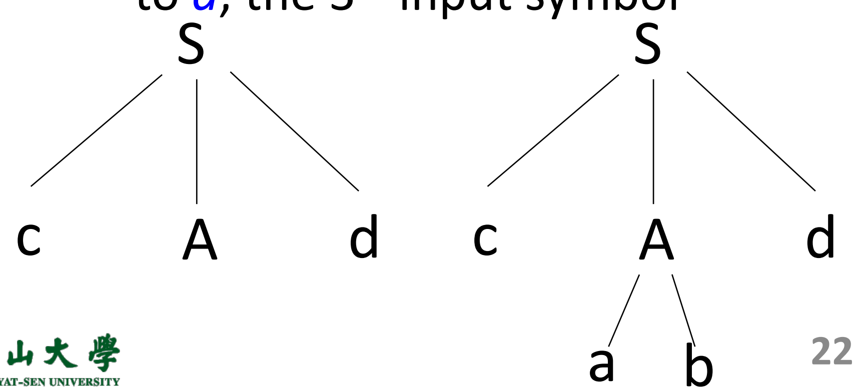


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - … …
  - The leftmost leaf, labeled *c*, matches the first symbol of *w*
    - So we advance the input pointer to *a* (i.e., the 2^{nd} symbol of *w*) and consider the next leaf *A*
  - Next, expand *A* using *A* → *ab*
    - Have a match for the 2^{nd} input symbol, *a*, so advance the input pointer to *d*, the 3^{rd} input symbol

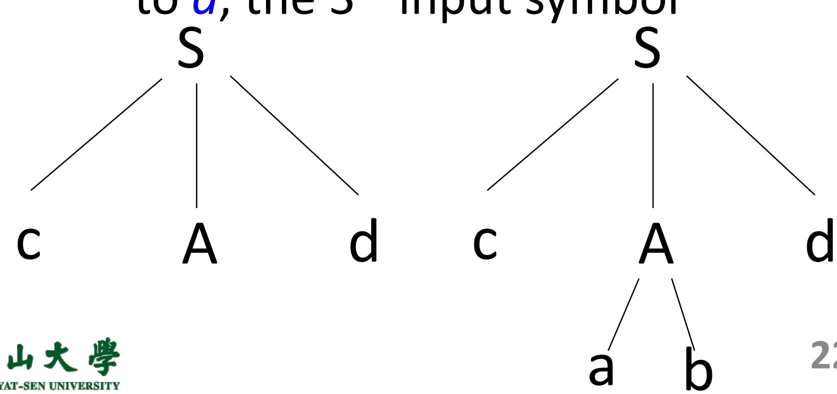


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - … …
  - The leftmost leaf, labeled *c*, matches the first symbol of *w*
    - So we advance the input pointer to *a* (i.e., the 2nd symbol of *w*) and consider the next leaf *A*
  - Next, expand *A* using *A* → *ab*
    - Have a match for the 2nd input symbol, *a*, so advance the input pointer to *d*, the 3rd input symbol

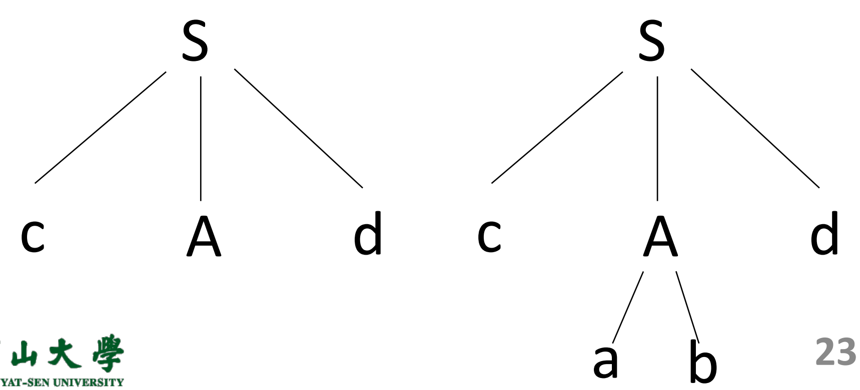


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - … …
  - *b* does not match *d*, report failure and go back to *A*
    - See whether there is another alternative for *A* that has not been tried
    - In going back to *A*, we must reset the input pointer as well

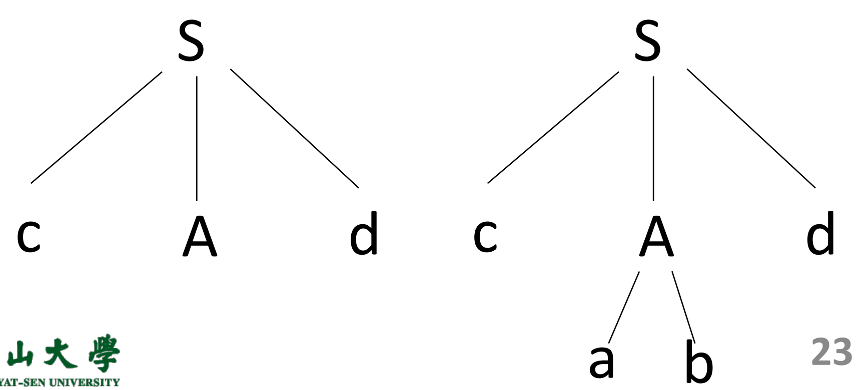


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - … …
  - *b* does not match *d*, report failure and go back to *A*
    - See whether there is another alternative for *A* that has not been tried
    - In going back to *A*, we must reset the input pointer as well

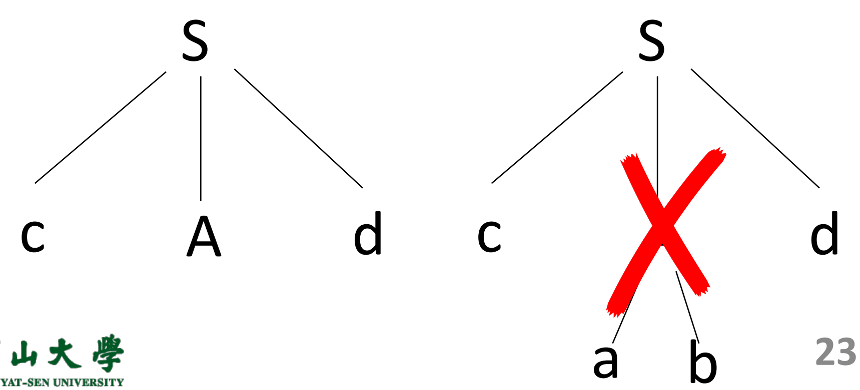


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - … …
  - *b* does not match *d*, report failure and go back to *A*
    - See whether there is another alternative for *A* that has not been tried
    - In going back to *A*, we must reset the input pointer as well

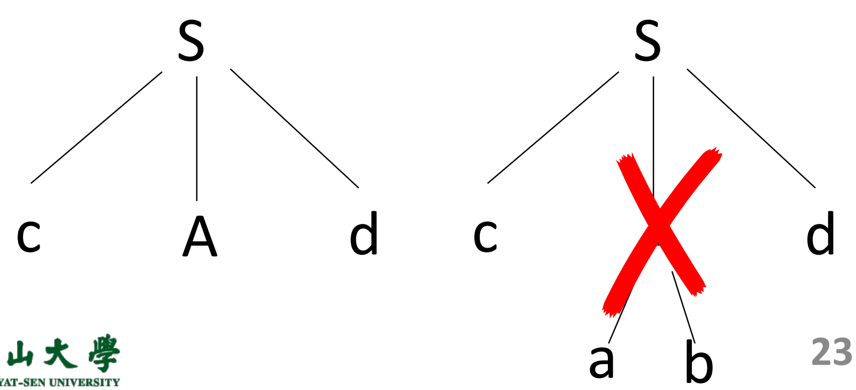


Example (cont.)

- Consider the grammar

  S → cAd A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - … …
  - *b* does not match *d*, report failure and go back to *A*
    - See whether there is another alternative for *A* that has not been tried
    - In going back to *A*, we must reset the input pointer as well
  - Leaf *a* matches the 2nd symbol of *w*, and leaf *d* matches the 3rd
  - We have produced a parse tree for *w*, we halt and success

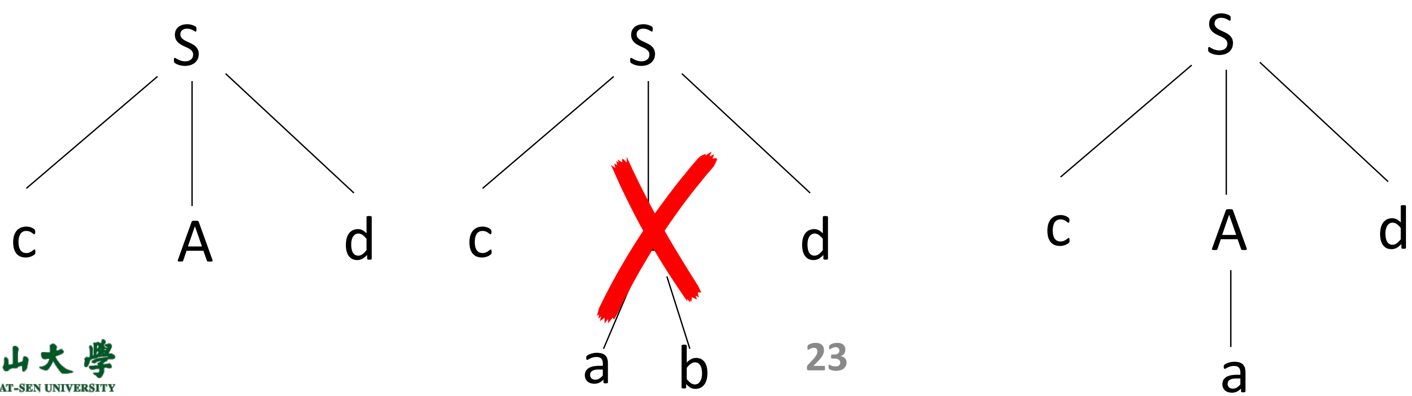


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - … …
  - *b* does not match *d*, report failure and go back to *A*
    - See whether there is another alternative for *A* that has not been tried
    - In going back to *A*, we must reset the input pointer as well
  - Leaf *a* matches the 2nd symbol of *w*, and leaf *d* matches the 3rd
  - We have produced a parse tree for *w*, we halt and success



Left Recursion Problem[左递归问题]

- Recursive descent doesn't work with left recursion
  - Right recursion is OK
- Why is left recursion[左递归] a problem?
  - For left recursive grammar

    A→Ab|c
  - We may repeatedly choose to apply A b

    A ⇒ A b ⇒ A b b ...
  - Sentential form can grow indefinitely w/o consuming input[句型无限增长而不消耗输入]
    - Non-terminal: expand, terminal: match
  - How do you know when to stop recursion and choose *c*?
- Rewrite the grammar so that it is right recursive[改为右递归]
  - Which expresses the same language[等价]

Left Recursion[左递归]

- A grammar is <u>left recursive</u> if
  - It has a nonterminal *A* such that there is a derivation $A \Rightarrow^{+} A\alpha$ for some string α
- Recursion types [直接和间接左递归]
  - **Immediate left recursion**, where there is a production $A \rightarrow A\alpha$
  - Non-immediate: left recursion involving derivation of 2+ steps

    $S \rightarrow Aa \mid b$

    $A \rightarrow Sd \mid \varepsilon$
  - $S \Rightarrow Aa \Rightarrow Sda$
- ☞ Algorithm to systematically eliminates left recursion from a grammar

Remove Left Recursion[消除左递归]

- Grammar: $A \rightarrow A\alpha \mid \beta$ ($\alpha \neq \beta$, β doesn't start with A)

  $A \Rightarrow A\alpha$

  $\Rightarrow A\alpha\alpha$

  …

  $\Rightarrow A\alpha\ldots\alpha\alpha$

  $\Rightarrow \beta\alpha\ldots\alpha\alpha$

  $r = \beta\alpha^*$

- Rewrite to:

  $A \rightarrow \beta A'$ // begins with β (A' is a new non-terminal)

  $A' \rightarrow \alpha A' \mid \varepsilon$ // A' is to produce a sequence of α

  $\Rightarrow \alpha\alpha A'$

  …

  $\Rightarrow \alpha\ldots\alpha A' \Rightarrow \alpha\ldots\alpha$

Remove Left Recursion (cont.)

- Grammar:

  $A \rightarrow A\alpha \mid \beta$

  to

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' \mid \varepsilon$

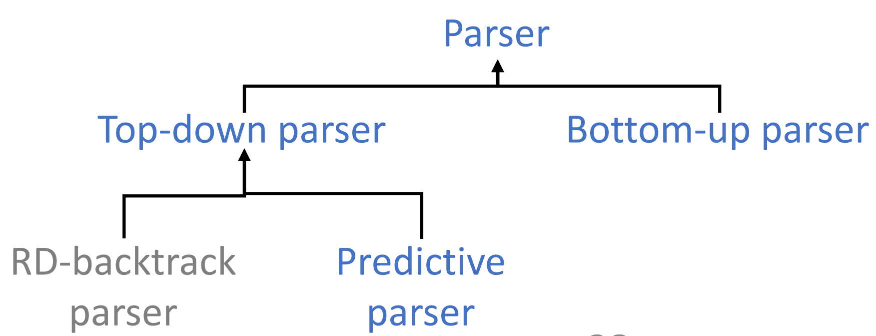
- $E \rightarrow E \underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$ ⇒ $E \rightarrow TE'$, $E' \rightarrow +TE' \mid \varepsilon$

- $T \rightarrow T \underbrace{*F}_{\alpha} \mid \underbrace{F}_{\beta}$ ⇒ $T \rightarrow FT'$, $T' \rightarrow *FT' \mid \varepsilon$

- $F \rightarrow (E) \mid id$ ⇒ $F \rightarrow (E) \mid id$

Summary of RD-backtrack[小结]

- **RD-backtrack** is a simple and general parsing strategy
  - Left-recursion must be eliminated first
    - Can be eliminated automatically using some algorithm
  - L(Recursive_descent) ≡ L(CFG) ≡ CFL
- However it is not popular because of **backtracking**
  - Backtracking requires re-parsing the same string
  - Which is inefficient (can take exponential time)
  - Also undoing semantic actions may be difficult
    - E.g. removing already added nodes in parse tree



Predictive Parsers[预测分析]

- In recursive descent with backtracking[有回溯]:
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- A parser with no backtracking[无回溯]: **predict** correct next production given next input terminal(s)?[以下面一些输入来预测]
  - If first terminal of every alternative production is **unique**, then parsing requires no backtracking[候选产生式首符号唯一]
  - If not unique, grammar cannot use predictive parsers[不唯一]

A→aBD | bBB
B→c | bce
D→d

parsing input "abced" requires no backtracking

?: 如果只往前看一个，那么next terminal其实就是current terminal，即要匹配的那个（注意backtrack是完全不看）