Positive Properties of Context-Free languages

class 12

1

Union

Context-free languages are closed under: **Union**

$$\left.\begin{array}{l} L_1 \text{ is context free} \\ L_2 \text{ is context free} \end{array}\right\} \Rightarrow L_1 \cup L_2 \text{ is context-free}$$

2

Example

Language	Grammar
$L_1 = \{a^n b^n\}$	$S_1 \to aS_1 b \mid \lambda$
$L_2 = \{ww^R\}$	$S_2 \to aS_2 a \mid bS_2 b \mid \lambda$

Union

$L = \{a^n b^n\} \cup \{ww^R\}$ $\qquad S \to S_1 \mid S_2$

3

In general:

For context-free languages $L_1,\ L_2$
with context-free grammars $G_1,\ G_2$
and start variables $S_1,\ S_2$

The grammar of the **union** $L_1 \cup L_2$
has new start variable S
and additional production $S \to S_1 \mid S_2$

4

Concatenation

Context-free languages are closed under: **Concatenation**

$$\left.\begin{array}{l} L_1 \text{ is context free} \\ L_2 \text{ is context free} \end{array}\right\} \Rightarrow L_1 L_2 \text{ is context-free}$$

5

Example

Language	Grammar
$L_1 = \{a^n b^n\}$	$S_1 \to aS_1 b \mid \lambda$
$L_2 = \{ww^R\}$	$S_2 \to aS_2 a \mid bS_2 b \mid \lambda$

Concatenation

$L = \{a^n b^n\}\{ww^R\}$ $\qquad S \to S_1 S_2$

6

In general:

For context-free languages $L_1, \ L_2$
with context-free grammars $G_1, \ G_2$
and start variables $S_1, \ S_2$

The grammar of the **concatenation** $L_1 L_2$
has new start variable S
and additional production $S \rightarrow S_1 S_2$

Star Operation

Context-free languages
are closed under: **Star-operation**

L is context free $\Rightarrow$ L^* is context-free

Example

Language	Grammar
$L = \{a^n b^n\}$	$S \rightarrow aSb \mid \lambda$

Star Operation

$L = \{a^n b^n\}^*$	$S_1 \rightarrow SS_1 \mid \lambda$

In general:

For context-free language L
with context-free grammar G
and start variable S

The grammar of the **star operation** L^*
has new start variable S_1
and additional production $S_1 \rightarrow SS_1 \mid \lambda$

Negative Properties of Context-Free Languages

Intersection

Context-free languages
are **not** closed under: **intersection**

L_1 is context free
L_2 is context free
$\Rightarrow$ $L_1 \cap L_2$ **not** necessarily context-free

Example

$L_1 = \{a^n b^n c^m\}$

Context-free:

$S \to AC$

$A \to aAb \mid \lambda$

$C \to cC \mid \lambda$

$L_2 = \{a^n b^m c^m\}$

Context-free:

$S \to AB$

$A \to aA \mid \lambda$

$B \to bBc \mid \lambda$

Intersection

$L_1 \cap L_2 = \{a^n b^n c^n\}$ NOT context-free

13

Complement

Context-free languages are **not** closed under: **complement**

L is context free $\Rightarrow$ $\overline{L}$ **not** necessarily context-free

14

Example

$L_1 = \{a^n b^n c^m\}$

Context-free:

$S \to AC$

$A \to aAb \mid \lambda$

$C \to cC \mid \lambda$

$L_2 = \{a^n b^m c^m\}$

Context-free:

$S \to AB$

$A \to aA \mid \lambda$

$B \to bBc \mid \lambda$

Complement

$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2 = \{a^n b^n c^n\}$

NOT context-free

15

Intersection of Context-free languages and Regular Languages

16

The intersection of
a context-free language and
a regular language
is a context-free language

L_1 context free

L_2 regular

$\Rightarrow$ $L_1 \cap L_2$ context-free

17

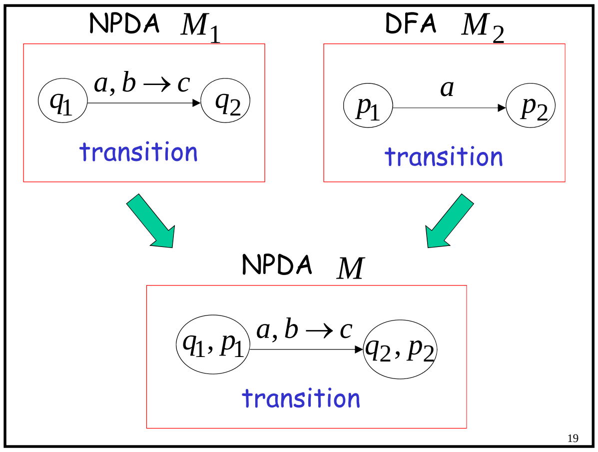
Machine M_1: NPDA for L_1, context-free

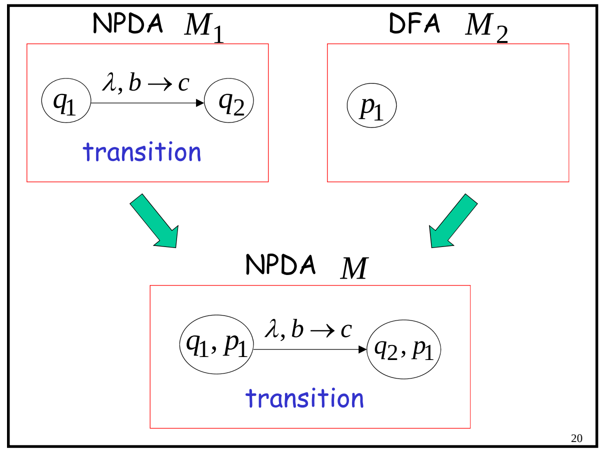
Machine M_2: DFA for L_2, regular

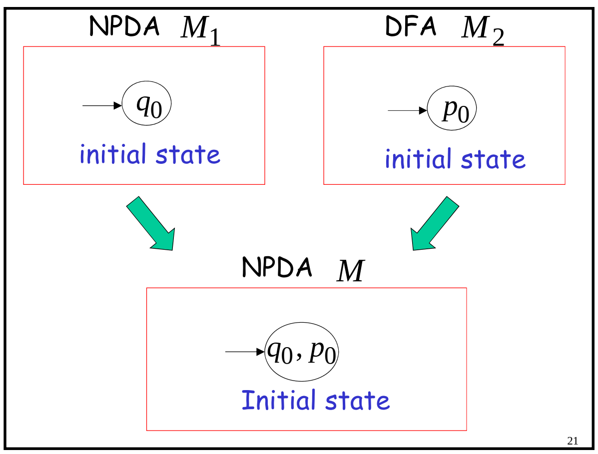
Construct a new NPDA machine M that accepts $L_1 \cap L_2$

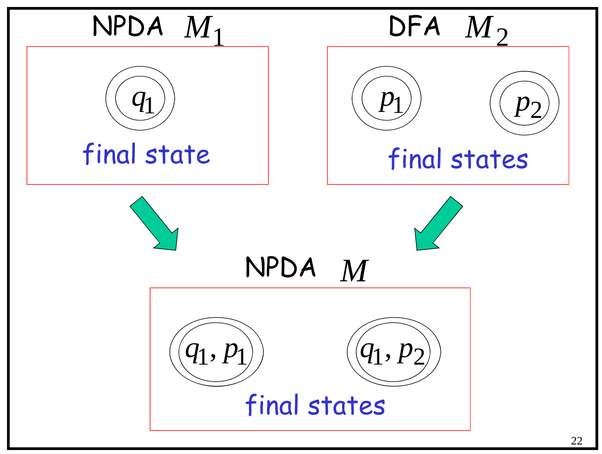
M simulates in parallel M_1 and M_2

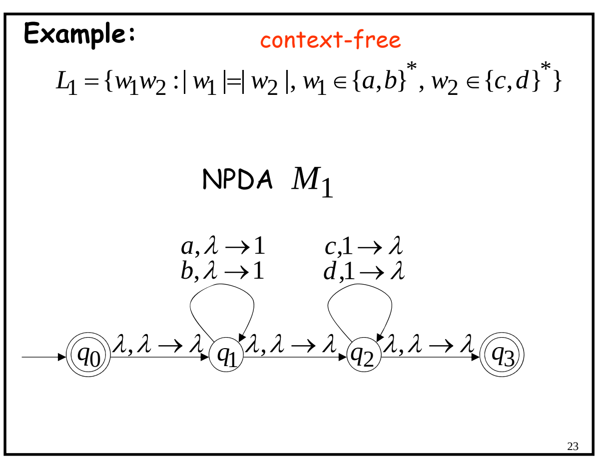
18

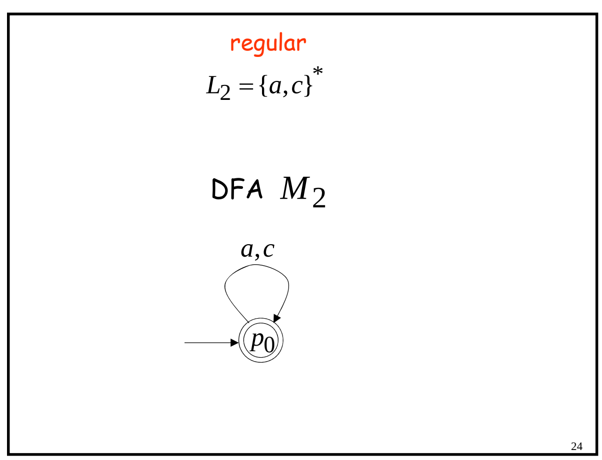

NPDA M_1
DFA M_2
$a, b \to c$
q_1
q_2
transition
a
p_1
p_2
transition
NPDA M
$a, b \to c$
q_1, p_1
q_2, p_2
transition
19


NPDA M_1
DFA M_2
$\lambda, b \to c$
q_1
q_2
transition
p_1
NPDA M
$\lambda, b \to c$
q_1, p_1
q_2, p_1
transition
20


NPDA M_1
DFA M_2
q_0
initial state
p_0
initial state
NPDA M
q_0, p_0
Initial state
21


NPDA M_1
DFA M_2
q_1
final state
p_1
p_2
final states
NPDA M
q_1, p_1
q_1, p_2
final states
22

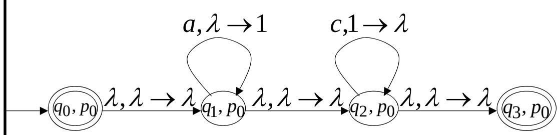

Example:
context-free
$L_1 = \{w_1 w_2 : |w_1| = |w_2|, w_1 \in \{a,b\}^*, w_2 \in \{c,d\}^*\}$
NPDA M_1
$a, \lambda \to 1$
$b, \lambda \to 1$
$c, 1 \to \lambda$
$d, 1 \to \lambda$
q_0
$\lambda, \lambda \to \lambda$
q_1
$\lambda, \lambda \to \lambda$
q_2
$\lambda, \lambda \to \lambda$
q_3
23


regular
$L_2 = \{a, c\}^*$
DFA M_2
a, c
p_0
24

context-free

Automaton for: $L_1 \cap L_2 = \{a^n c^n : n \geq 0\}$

NPDA M



In General:

M simulates in parallel M_1 and M_2

M accepts string w if and only if

M_1 accepts string w and

M_2 accepts string w

$$L(M) = L(M_1) \cap L(M_2)$$

Therefore:

M is NPDA

$\Downarrow$

$L(M_1) \cap L(M_2)$ is context-free

$\Downarrow$

$L_1 \cap L_2$ is context-free

Applications of Regular Closure

The intersection of
a context-free language and
a regular language
is a context-free language

L_1 context free
L_2 regular
$\Rightarrow$ $L_1 \cap L_2$ context-free

Regular Closure

An Application of Regular Closure

Prove that: $L = \{a^n b^n : n \neq 100, n \geq 0\}$

is context-free

We know:

$\{a^n b^n : n \geq 0\}$ is context-free

We also know:

$L_1 = \{a^{100} b^{100}\}$ is regular

$\Downarrow$

$\overline{L_1} = \{(a+b)^*\} - \{a^{100} b^{100}\}$ is regular

$\{a^n b^n\}$ context-free

$\overline{L_1} = \{(a+b)^*\} - \{a^{100} b^{100}\}$ regular

(regular closure) $\{a^n b^n\} \cap \overline{L_1}$ context-free

$\Downarrow$

$\{a^n b^n\} \cap \overline{L_1} = \{a^n b^n : \ n \neq 100, n \geq 0\} = L$

is context-free

Another Application of Regular Closure

Prove that: $L = \{w : \ n_a = n_b = n_c\}$

is **not** context-free

If $L = \{w : \ n_a = n_b = n_c\}$ is context-free

(regular closure)

Then $L \cap \{a^* b^* c^*\} = \{a^n b^n c^n\}$

context-free regular context-free

Impossible!!!

Therefore, L is **not** context free

Decidable Properties
of
Context-Free Languages

Membership Question:

for context-free grammar G
find if string $w \in L(G)$

Membership Algorithms: Parsers

- Exhaustive search parser
- **CYK** parsing algorithm

37

Empty Language Question:

for context-free grammar G
find if $L(G) = \emptyset$

Algorithm:

1. Remove useless variables
2. Check if start variable S is useless

38

Infinite Language Question:

for context-free grammar G
find if $L(G)$ is infinite

Algorithm:

1. Remove useless variables
2. Remove unit and λ productions
3. Create dependency graph for variables
4. If there is a loop in the dependency graph then the language is infinite

39

Example: $S \to AB$

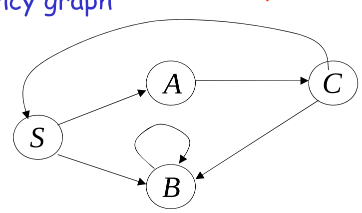
$A \to aCb \mid a$

$B \to bB \mid bb$

$C \to cBS$

Dependency graph

Infinite language



40

$$S \to AB$$
$$A \to aCb \mid a$$
$$B \to bB \mid bb$$
$$C \to cBS$$

$$S \Rightarrow AB \Rightarrow aCbB \Rightarrow acBSbB \Rightarrow acbbSbbb$$

$$S \overset{*}{\Rightarrow} acbbSbbb \overset{*}{\Rightarrow} (acbb)^2 S (bbb)^2$$

$$\overset{*}{\Rightarrow} (acbb)^i S (bbb)^i$$

41

YACC

Yet Another Compiler Compiler

42

Yacc is a parser generator

Input: A Grammar

Output: A parser for the grammar

Reminder: a parser finds derivations

Example grammar:

```
expr -> ( expr )
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | - expr
      | INT
      ;
```

The yacc code:

```
expr : '(' expr ')'
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | - expr
     | INT
     ;
```


Exampe Input:

10 * 3 + 4

Yacc Derivation:

expr => expr + expr => expr * expr + expr
=> 10*3 + 4

Resolving Ambiguities

```
%left '+', '-'
%left '*', '/'
%left UMINUS
%%

expr : '(' expr ')'
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '-' expr      %prec UMINUS
     | INT
     ;
```


Actions

```
%left '+', '-'
%left '*', '/'
%left UMINUS
%%

expr : '(' expr ')'      {$$ = $2;}
     | expr '+' expr     {$$ = $1 + $3;}
     | expr '-' expr     {$$ = $1 - $3;}
     | expr '*' expr     {$$ = $1 * $3;}
     | expr '/' expr     {$$ = $1 / $3;}
     | '-' expr     %prec UMINUS    {$$ = -$2;}
     | INT               {$$ = $1;}
     ;
```


A Complete Yacc program

```
%union{
  int int_val;
}

%left '+', '-'
%left '*', '/'
%left UMINUS

%token <int_val> INT
%type <int_val> expr

%start program

%%
```

```
program : expr        {printf("Expr value = %d \n", $1);}
        | error       {printf("YACC: syntax error near line %d \n", linenum);
                       abort();}
        ;

expr : '(' expr ')'     {$$ = $2;}
     | expr '+' expr    {$$ = $1 + $3;}
     | expr '-' expr    {$$ = $1 - $3;}
     | expr '*' expr    {$$ = $1 * $3;}
     | expr '/' expr    {$$ = $1 / $3;}
     | '-' expr         %prec UMINUS  {$$ = -$2;}
     | INT              {$$ = $1;}
     ;

%%

#include "lex.yy.c"
```

49

Execution Example

Input: 10 + 20*(3 - 4 + 25)

Output: Expr value = 490

50

The Lex Code

```
%{
int linenum=1;
int temp_int;
%}
%%

\n      {linenum++;}

[\t ]        /* skip spaces */;
\/\/[^\n]*    /* ignore comments */;

"+"     {return '+';}
"-"     {return '-';}
"*"     {return '*';}
"/"     {return '/';}
")"     {return ')';}
"("     {return '(';}
```

51

```
[0-9]+   {sscanf(yytext, "%d", &temp_int);
          yylval.int_val = temp_int;
          return INT;}

.  {printf("LEX: unknown input string found in line %d \n", linenum);
   abort();}
```

52

Compiling:

```
yacc YaccFile
lex LexFile
cc y.tab.c -ly -ll -o myparser
```

Executable: myparser

53

Another Yacc Program

```
%union{
  int int_val;
}

%left '+', '-'
%left '*', '/'
%left UMINUS

%token <int_val> INT
%type <int_val> expr

%start program

%%
```

54

```
program : stmt_list
      | error        {printf("YACC: syntax error near line %d \n", linenum);
                      abort();}
      ;

stmt_list : stmt_list stmt
       | stmt
       ;

stmt : expr ';'    {printf("Expr value = %d \n", $1);}
    ;
```



```
expr : '(' expr ')'     {$$ = $2;}
   | expr '+' expr    {$$ = $1 + $3;}
   | expr '-' expr    {$$ = $1 - $3;}
   | expr '*' expr    {$$ = $1 * $3;}
   | expr '/' expr    {$$ = $1 / $3;}
   | '-' expr         %prec UMINUS {$$ = -$2;}
   | INT              {$$ = $1;}
   ;

%%

#include "lex.yy.c"
```


Execution Example

Input:

```
10 + 20*(30 -67) / 4;

34 * 35 - 123 + -001;

17*8/6;
```

Output:

```
Expr value = -175
Expr value = 1066
Expr value = 22
```


Lex Code

```
%{
int linenum=1;
int temp_int;
%}
%%

\n     {linenum++;}

[\t ]       /* skip spaces */;
\/\/[^\n]*   /* ignore comments */;
```



```
"+"    {return '+';}
"-"    {return '-';}
"*"    {return '*';}
"/"    {return '/';}
")"    {return ')';}
"("    {return '(';}
";"    {return ';';}

[0-9]+   {sscanf(yytext, "%d", &temp_int);
         yylval.int_val = temp_int;
         return INT;}

.  {printf("LEX: unknown input string found in line %d \n", linenum);
   abort();}
```


Another Yacc Program

```
%union{
  int int_val;
  char *str_val;
}

%left '+', '-'
%left '*', '/'
%left UMINUS

%token PRINT
%token NEWLINE
%token <str_val> STRING
%token <int_val> INT
%type <int_val> expr

%start program
%%
```

```
program : stmt_list
        | error       {printf("YACC: syntax error near line %d \n", linenum);
                       abort();}
        ;

stmt_list : stmt_list stmt
          | stmt
          ;

stmt : expr ';'                    {printf("expression found\n");}
     | PRINT expr ';'              {printf("%d", $2);}
     | PRINT STRING ';'            {printf("%s", $2);}
     | PRINT NEWLINE ';'           {printf("\n");}
     ;
```



```
expr : '(' expr ')'     {$$ = $2;}
     | expr '+' expr    {$$ = $1 + $3;}
     | expr '-' expr    {$$ = $1 - $3;}
     | expr '*' expr    {$$ = $1 * $3;}
     | expr '/' expr    {$$ = $1 / $3;}
     | '-' expr         %prec UMINUS {$$ = -$2;}
     | INT              {$$ = $1;}
     ;

%%

#include "lex.yy.c"
```


Execution Example

Input:

```
print "The value of expression 123 * 25 is ";
print 123 * 25;
print newline;
10 + 5 * 8;
print "end of program";
print newline;
```

Output:

```
The value of expression 123 * 25 is 3075
expression found
end of program
```


Lex Code

```
%{
int linenum=1;
int temp_int;
char temp_str[200];
%}
%%

\n      {linenum++;}

[\t ]         /* skip spaces */;
\/\/[^\n]*     /* ignore comments */;
```



```
"+"       {return '+';}
"-"       {return '-';}
"*"       {return '*';}
"/"       {return '/';}
")"       {return ')';}
"("       {return '(';}
";"       {return ';';}
"print"    {return PRINT;}
"newline"  {return NEWLINE;}
```



```
[0-9]+   {sscanf(yytext, "%d", &temp_int);
          yylval.int_val = temp_int;
          return INT;}

\"[^"\n]*\" {strncpy(temp_str, &(yytext[1]), strlen(yytext)-2);
             temp_str[strlen(yytext)-2] = (char) 0;
             yylval.str_val = temp_str;
             return STRING;}

.  {printf("LEX: unknown input string found in line %d \n", linenum);
   abort();}
```