補充資料 I
(Regular Grammar)

資科系
林偉川

語言的定義

- 任一上下文無關的文法 (Type 2)的語言，可訂出其相對的文法來描述該語言
- 文法即代表語言本身的特性
- Regular grammar ➔ Regular Language
- Language L 被稱為regular的前提為找到一個或多個特定有限接受機器(deterministic finite accepter簡稱DFA) M，使得此機器可產生此語言

2

文法定義

G=(V,T,S,P) , where

V : a finite set of variables(非終端符號)

T : a finite set of terminal symbols(終端符號)

S ∈V is a special symbol called起始符號

P is a finite set of productions(文法產生規則)

一個文法產生規則有一個左手邊(LHS)[*non-terminal*]及一個右手邊 (RHS), 而右手邊包括 *終端及非終端*符號

一個文法產生規則是*遞迴* 為其 LHS出現在其RHS

<ident_list> → identifier | identifier, <ident_list>

3

文法推導範例

G={ {program,stmt_list,stmt,var,expression}, {;,=,+,-,a,b,c,d}, program, P}

P1. <program> → <stmt_list> (<program>是起始符號)

P2. <stmt_list> → <stmt> | <stmt> ; <stmt_list>

P3. <stmt> → <var> = <expression>

P4. <var> → a | b | c | d | e

P5. <expression> → <var> + <expression> | <var> - <expression> | <var>

An example leftmost derivation

<program> ⇒ <stmt_list> ⇒ <stmt> ; <stmt_lsit> (1,2)

⇒ <var> = <expression>; <stmt_list> (3)

⇒ a = <expression>; <stmt_list> (4)

4

文法推導範例

⇒ a = <var> + <var>; <stmt_list> (5)
⇒ a = b + <var>; <stmt_list> (4)
⇒ a = b + c; <stmt_list> (4)
⇒ a = b + c; <stmt> (2)
⇒ a = b + c; <var> = <expression> (3)
⇒ a = b + c; b = <expression> (4)
⇒ a = b + c; b = <var> (5)
⇒ a = b + c; b = c (4)

How about a=b+c-d+e ?

每一個推導過程所產生的字串稱為***sentential form***

一個被接受字串為一個只包括終端符號的**sentential form**

5

文法練習

- B.N.F.文法規則定義如下：

G={{S,A,B,C},{a,b,c},S,P}

P1.<S>➔<A><B><C>

P2. <A>➔ a<A> | a

P3. <B>➔ b<B> | b

P4. <C>➔ c<C> | c

請問此文法所產生的語言特性為何？

6

文法練習

- B.N.F.文法規則定義如下：

G={{S,A,B},{a,b},S,P}

P1.<S>::=<A>a<B>b

P2. <A>::= <A>b | b

P3. <B> ::= a<B> | a

請問此文法可產生下列哪些語言？

(1) baab (2) bbbab (3) bbaaaaa (4) bbaab

7

文法練習

- B.N.F.文法規則定義如下：

G={{S,A,B},{a,c,b,d},S,P}

P1.<S>::=a<S>c<B> | <A> | b

P2. <A>::= c<A> | c

P3. <B> ::= <A> | d

請問此文法可產生下列哪些語言？

(1) abcd (2) acccbd (3) acccbcc (4) acd (5) accc

8

文法練習

- B.N.F.文法規則定義如下：

G={{assign,id,expr}, {:=,a,b,c,+,*,(,)}, assign, P}

1.<assign>::= <id> := <expr>

2. <id>::= a | b | c

3. <expr> ::= <id>+<expr> | <id> | <id>*<expr> | (<expr>)

請畫出下列語言的剖析樹？

(1) a:=b*(a+c) (2) a:=a*(b+(c*a)) (3) b:=c*(a*c+b) (4) a:=a*(b+(c))

9

BNF v.s. EBNF

BNF:

<expr> →<expr> + <term> | <expr> - <term> | <term>

<term>→<term> * <factor> | <term> / <factor> | <factor>

EBNF:

<expr>→<term> {(+ | -) <term>}

<term> →<factor> {(* | /) <factor>}

10

DFA Definition(Regular Grammar)

M=(Q, Σ, δ, q_0, F) where

Q：a finite set of internal states

Σ ： a finite set of input alphabets

δ ： Q * Σ → Q is a transition function

q_0 ∈ Q is the initial state

F ⊆ Q is a set of finite states

11

範例

文法G=({S,A},{a,b},S, P), where P are given by

S → aAa

A → aA | bA | λ | is logical OR for production.

此語言接受開頭及結尾為a的字串!

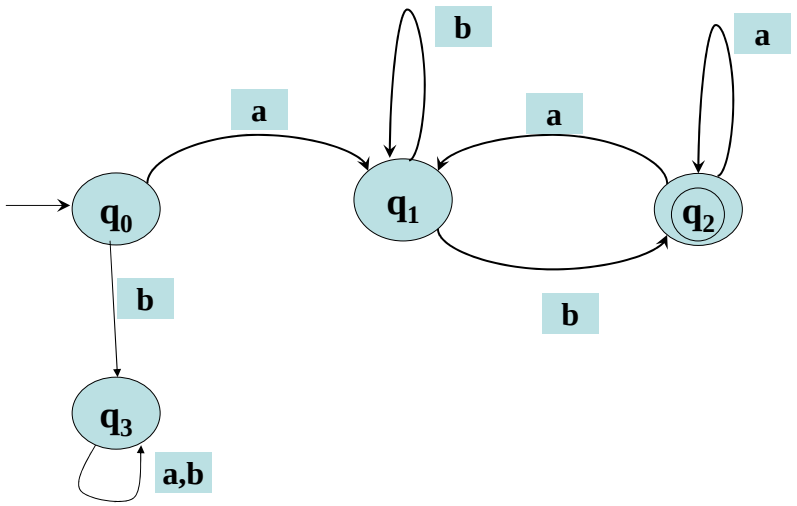
推導過程： S ⇒ aAa ⇒ aaAa ⇒ aabAa ⇒ aaba

Grammar produces a language result:

L(G)={ awa: w ∈ {a,b}* **}**

12

DFA範例



13

DFA範例

M=({q_0,q_1,q_2,q_3}, {a,b}, δ, q_0, {q_2})

Where δ is given by

$\delta(q_0,a)=q_1$ ， $\delta(q_0,b)=q_3$ ，

$\delta(q_1,a)=q_2$ ， $\delta(q_1,b)=q_1$ ，

$\delta(q_2,a)=q_2$ ， $\delta(q_2,b)=q_1$ ，

$\delta(q_3,a)=q_3$ ， $\delta(q_3,b)=q_3$ 。

Input symbol / Current State	a	b
q_0	q_1	Q_3
q_1	q_2	Q_1
q_2	q_2	Q_1
q_3	q_3	Q_3

DFA produces a language result:

L(G)={ awa: w ∈ {a,b}* }

範例

G=({S},{a,b}, S, P), where P are given by

$S \rightarrow aSb \mid \lambda$ (recursive grammar rule)

| is logical OR for production.

Derivation processes:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

Grammar produces a language result:

$L(G)=\{ a^n b^n : n >= 0 \}$

15

範例

G=({S,A},{a,b}, S, P), where P are given by

$S \rightarrow Ab$

$A \rightarrow aAb \mid \lambda$

Grammar produces a language result:

$L(G)=\{ a^n b^{n+1} : n >= 0 \}$

16

範例

G=({S},{a,b}, S, P), where P are given by

$S \rightarrow SS \mid \lambda \mid aSb \mid bSa$

Grammar produces a language result:

$L(G)=\{ w: n_a(w)=n_b(w) \}$

17

範例

G=({S,A},{a,b}, S, P), where P are given by

$S \rightarrow aAb$

$A \rightarrow aAb \mid \lambda$

Grammar produces a language result:

$L(G)=\{ a^n b^n: n > 0 \}$

18

DFA範例

M=({q_0,q_1,q_2}, {0,1}, δ, q_0, {q_1})

Where δ is given by

$\delta(q_0,0)=q_0$ ， $\delta(q_0,1)=q_1$ ，

$\delta(q_1,0)=q_0$ ， $\delta(q_1,1)=q_2$ ，

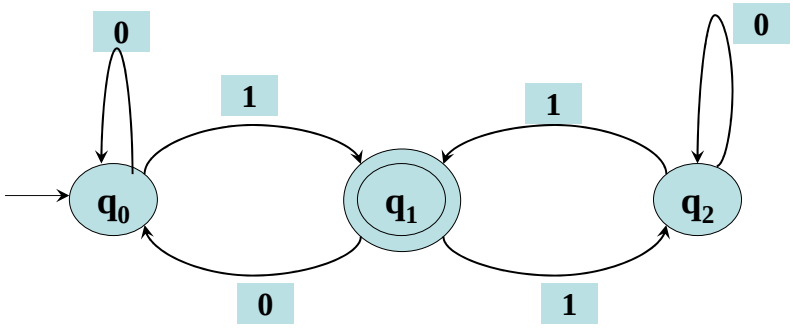
$\delta(q_2,0)=q_2$ ， $\delta(q_2,1)=q_1$ ，

Input symbol / Current State	0	1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_2	q_1

DFA produces a language result:

L(DFA)={ 字串結尾為奇數個1 }

DFA範例



20

DFA 程式範例

```
State=0;
While (字串不結束) {
switch (state) {
Case 0:
   if (ch == ‘1’) state=1; break;
Case 1:
   if (ch == ‘1’) state=2;
   if (ch == ‘0’) state=0; break;
…..
}}
If (state == 1) printf(“輸入字串可被處理\n”)；
```

21

```
#include <stdio.h>
void main() {
  int state=0;  char c;  clrscr();
  while ((c=getchar()) != '\n') {
    switch (state) {
    case 0:
      if (ch == ‘0’) state=0;
      if (ch == ‘1’) state=1;
      break;
    case 1:
       if (ch == ‘1’) state=2;
       if (ch == ‘0’) state=0;
       break;
    case 2:
      if (ch == ‘1’) state=1;
      if (ch == ‘0’) state=0;
      break;
    }
   printf("%c stat=%d\n", c,state);
 }
```

```
if (state == 1)
   printf("輸入字串可被DFA處理\n");
else
  printf(“輸入字串不可被DFA處理\n");
}
```

DFA and Regular Grammar範例

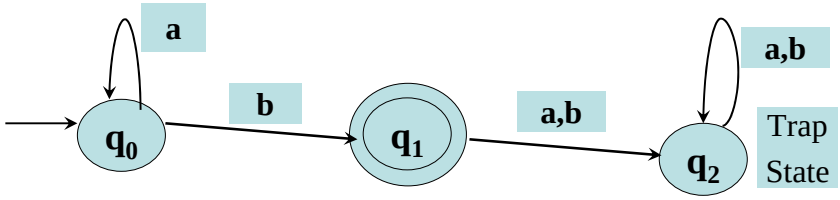
G=({S,A},{a,b}, S, P), where P are given by

$S \rightarrow Ab$

$A \rightarrow aA \mid \lambda$

Grammar produces a language result:

$L(G)=\{ a^n b: n >= 0 \}$



DFA and CFG範例

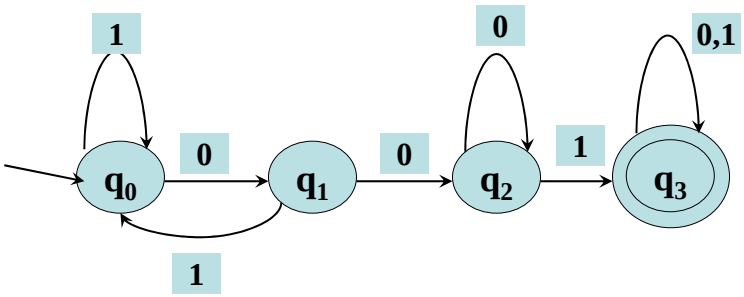
Input symbol / Current State	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

24

DFA and CFG範例

Find a DFA accept all strings on {0,1}

L(DFA)={字串包含子字串 001}？



25

```
while ((c=na[i++]) != '\0') {
    switch (state) {
    case 0: if (c == '0') state=1;      break;
    case 1: if (c == '1') state=0;    if (c == '0') state=2;   break;
    case 2: if (c == '1') state=3;   break;
    case 3:  accept=1;   break;
    }
}
if (accept == 0) printf("%s is not accept for substring 001\n",na);
else printf("%s is accept for substring 001\n",na);
```

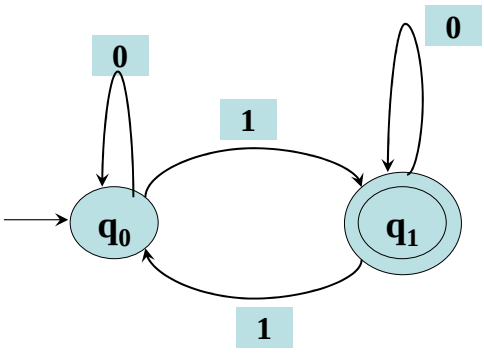
26

DFA and CFG範例

- Find a DFA produces a language result:
  - **a. L(DFA)={ 字串包含奇數個1 } ➔ {0,1}**
  - **b. L(DFA)={ awa: w** ∈ {a,b}* **} ➔ {a,b}**
  - **c. L(DFA)={**字串結尾為 001} **➔ {0,1}**
  - **d. L(DFA)={ aw_1aaw_2a: w_1及w_2 ∈ {a,b}* }**
  - **e. L(DFA)={**字串包含至少1個1} **➔ {0,1}**
  - **f. L(DFA)={**字串包含不超過3個1} **➔ {0,1}**

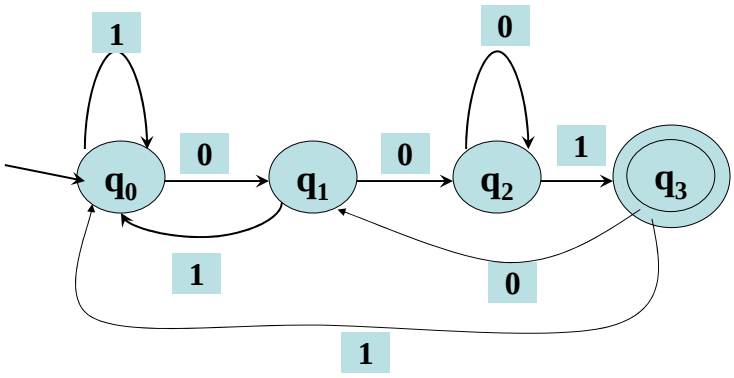
27

DFA範例字串包含奇數個1



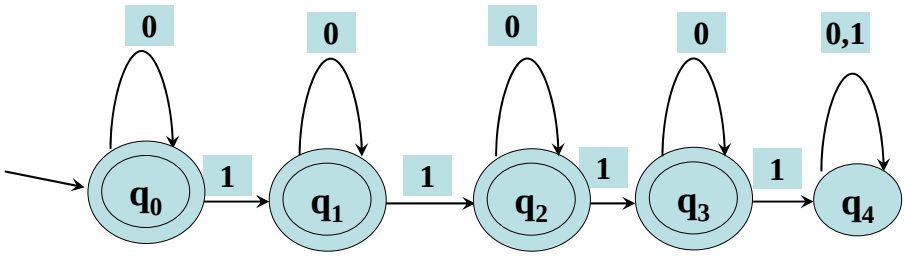
28

DFA範例字串字串結尾為 001



29

DFA範例字串包含不超過3個1



30

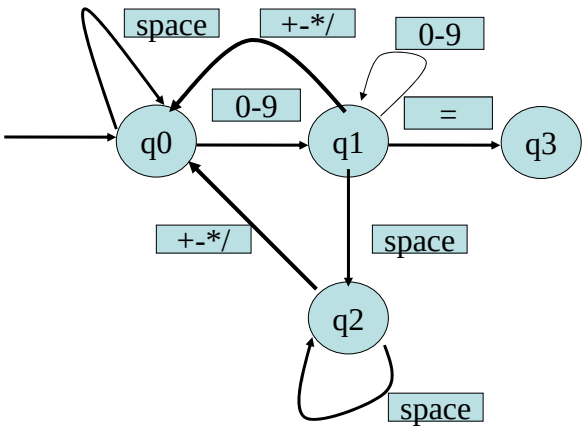
DFA and CFG範例

請找一DFA, 能判斷一運算式的對錯？
(以 = 為 final state) [至少需8個states]
Ex. 12 + 23 - 45 * 67=
[不可以接受 12 =]

A = 12 + ((23 - 45) * 67);
A0 = (12 + 23) - 45 * 67 ;

31

錯誤的例子



32

Recursive Descent Parsing(RDP)

- RDP is a 以文法為基礎的由上到下的文法剖析器
- 文法剖析過程是建構一顆自輸入字串推導過程的剖析樹
- 每一個非終端符號對應一個副程式來處理，此副程式剖析非終端符號所產生所有的 sentential forms
- RDP副程式是直接由文法規則產生的
- 文法剖析器首先呼叫字集分析器(lexical analyzer)對保留字及變數作切割及分類
- 給與一個輸入字串，它可自剖析樹的樹根之非終端符號，一直推導到最後的所有樹葉的組合，即為該字串

33

字集分析器(lexical analyzer)

- 語言的最低層的單位稱為lexemes，字集分析器即是分析一個語言的過程，將其組成的基本單位做有效的分類切割成為Token
- index=2*count+17;
- lexical();
- Symbol table

lexemes	Token
index	變數(1) int
=	等號(3)
2	常數(2)
*	乘號(4)
count	變數(1) byte
+	加號(4)
17	常數(2)
;	分號(5)

34

字集分析器之實作

```
int lexical() {
  switch (charClass) {
    case LETTER: //變數
      addChar();    getChar();
      while (charClass == LETTER || charClass == DIGIT) {
        addChar(); getChar();
      }
      return lookup(lexeme);  break;
    case DIGIT: //常數
      addChar(); getChar();
      while (charClass == DIGIT) { addChar(); getChar();   }
      return 2; break;
  }  /* End of switch */
}  /* End of function lex */
```

35

原始文法

- BNF

1. <expr> →<expr>+<term> | <expr>-<term> | <term>
2. <term> →<term>*<factor> | <term>/<factor> | <facotr>
3. <factor> →(<expr>) | <id>
4. <id> →<alpha><digit> | <alpha>
5. <alpha> → A | B | … | Z | a | b | … | z
6. <digit> → 0 | 1 | … | 9
7. <assign> → <id> = <expr>

36

EBNF文法規則

1. <expr> →<term> {(+ | -) <term>}
2. <term> →<factor> {(* | /) <factor>}
3. <factor> →<id> | (<expr>)
4. <id> →<alpha><digit>
5. <alpha> → A | B | … | Z | a | b | … | z
6. <digit> → 0 | 1 | … | 9
7. <assign> → <id> = <expr>

37

RDP副程式

```
void assign() {// 7.<assign> → <id> := <expr>
  id();  /* parse the first factor*/
 while (next_token ==‘=’) {
   lexical();  /* 從輸入取得下一個lexeme */
   expr();  /* parse the next term */
 }
}
```

38

RDP副程式

```
void digit() {// 6.<digit> → 0 | 1 | … | 9
  if (next_token ==‘0’|| …) lexical();
}
```

39

RDP副程式

```
void alpha() {//5.<alpha> → A | B | … | Z | a | b
  | … | z
 if (next_token ==‘A’|| …) lexical();
}
```

40

RDP副程式

```
void id() {// 4.<id> →<alpha><digit>
 alpha();  /* parse the first factor*/
 digit();
}
```

41

RDP副程式

```
void factor() { // 3. <factor> →<id> | ( <expr> )
if (next_token == id_code) { lexical(); return;}
else if (next_token == '(') {
       lexical(); expr();
       if (next_token == ')') { lexical(); return; }
       else error(); // 不能只顯示error，還得繼續執行
      }
      else error();  //不能只顯示error，還得繼續執行
}
```

42

RDP副程式

```
void term(){ //2. <term> →<factor> {(* | /)
   <factor>}
  factor();  /* parse the first factor*/
  while (next_token == '*'|| next_token == '/') {
    lexical(); /* 從輸入取得下一個lexeme */
    factor();  /* parse the next factor */
  }
}
```

43

RDP副程式

```
void expr() { //1.<expr> →<term> {(+ | -) <term>}
 term();  /* parse the first factor*/
 while (next_token =='+'|| next_token == '-') {
   lexical();  /* 從輸入取得下一個lexeme */
   term();  /* parse the next term */
 }
}
```

44

RDP副程式

```
void error() {
  printf(“some errors are found!!!\n”);
}
```

45