Parsing

EXPRESSION ABBREVIATIONS

Regular expression :

digits=[0-9]+

sum=(digits "+")*digits

=>defines sums of the form: **28+301+9.**

Consider :

digits=[0-9]+

sum=expr "+" expr

expr="(" sum ")"|digits

=>defines expressions of the form:

(109+23)   61   (1+(250+3))

- But it is impossible for a finite automaton to recognize balanced parentheses(because a machine with ***N*** states cannot remember a parenthesis-nesting depth greater than ***N***), so clearly ***sum*** and ***expr*** cannot be regular expressions**.**

HOW TO IMPLEMENT REGULAR-EXPRESSION ABBREVIATIONS?

The Answer:

- The right-hand-side ([0-9]+) is simply substituted for ***digits*** wherever it appears in regular expressions , ***before*** translation to a finite automaton**.**
- Substitute ***sum*** into ***expr*****,** yielding
  - ***expr=“(“ expr”+”expr”)”|digits***
- Now an attempt to substitute ***expr*** into itself leads to
  - ***expr=“(“(“(“ expr”+”expr”)”|digits)”+”expr”)”|digits***

CONTEXT-FREE GRAMMARS

A context-free grammar has a set of productions of the form:

symbol ->symbol symbol … symbol

Where

- **There are zero or more symbols on the right-hand side.**
- **Each symbol is either terminal, or non-terminal.**
- **No token can ever appear on the left-hand side of some production.**
- **One of non-terminal is distinguished as the start symbol of the grammar.**

CONTEXT-FREE GRAMMARS

- Derivations
  - Leftmost or Rightmost;
- Example for leftmost derivation

<u>S</u>

<u>S</u>;S

id:=<u>E</u>;S

id:=num;<u>S</u>

id:=num;id:=<u>E</u>

id:=num;id:=<u>E</u>+E

.

.

Ambiguous Grammar(1/2)

- 範例

有一文法 **G :E->E+E|E*E|(E)|-E|id**

=>id+id*id 產生兩種Leftmost Derivation

E=>E+E		E=>E*E
=>id+E		=>E+E*E
=>id+E*E		=>id+E*E
=>id+id*E	***Parsing Tree***	=>id+id*E
=>id+id*id		=>id+id*id

Ambiguous Grammar(2/2)

- 解決方法
  - 指明運算子是採左邊結合或右邊結合的方式

    若為左邊結合=>

    E=>E*E
    =>E+E*E
    =>id+E*E
    =>id+id*E
    =>id+id*id

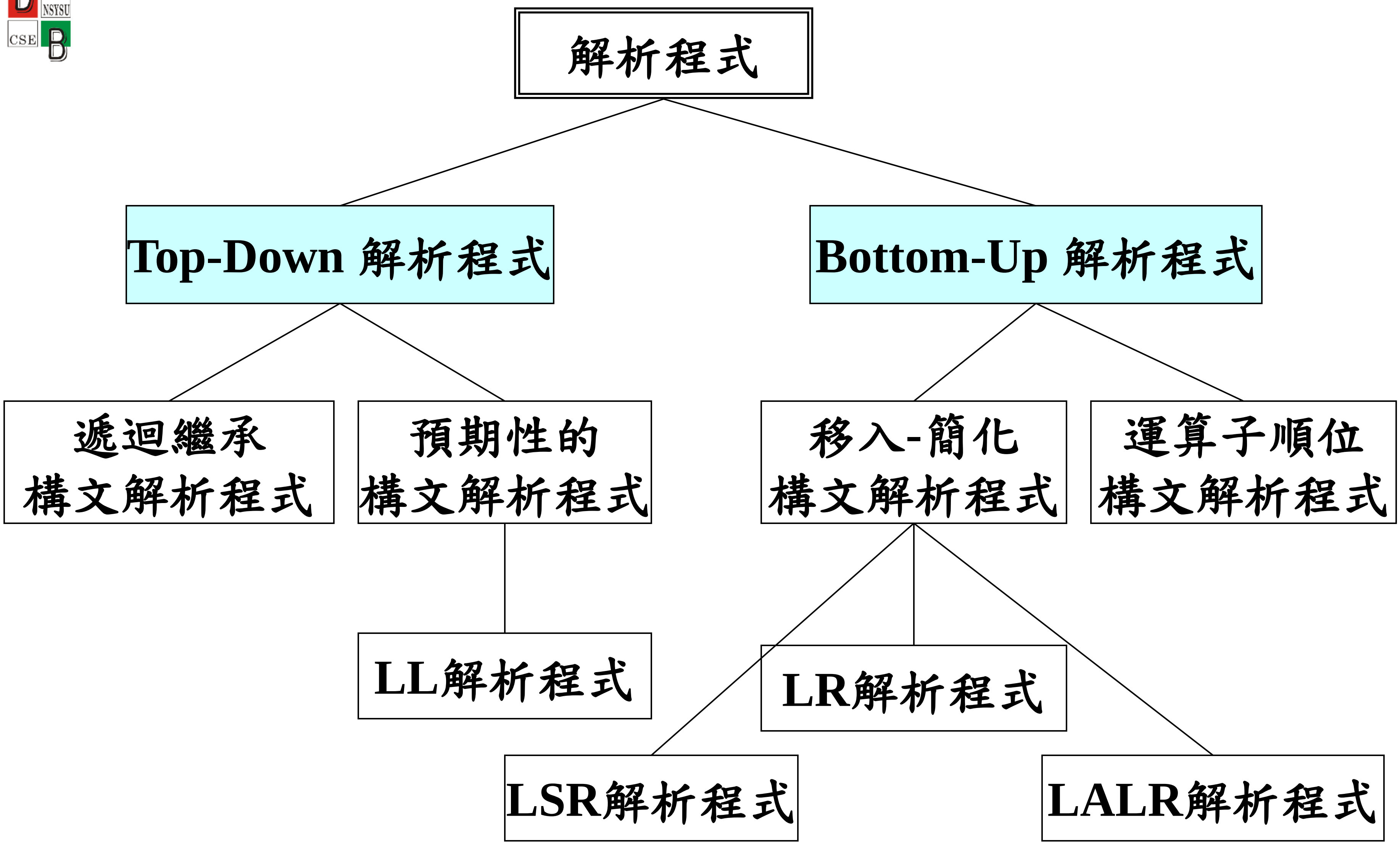
  - 定義運算子的順位

    若 **operator * > operator + =>**

    E=>E+E
    =>id+E
    =>id+E*E
    =>id+id*E
    =>id+id*id

剖析(Parsing)

- 方法
  - Top-Down Parser
    - 遞迴繼承構文解析程式(Recursive Descent Parser)
    - 預期性的構文解析程式(Predictive Parser)
  - Bottom-Up Parser
    - 移入-簡化構文解析程式(Shift-Reduce Parser)
    - 運算子順位構文解析程式(Operator Precedence Parser)


解析程式
Top-Down 解析程式
Bottom-Up 解析程式
遞迴繼承
構文解析程式
預期性的
構文解析程式
移入-簡化
構文解析程式
運算子順位
構文解析程式
LL解析程式
LR解析程式
LSR解析程式
LALR解析程式

遞迴繼承構文解析程式(1/8)

1. 一般Top-Down Parsing 常需要做數次的回頭掃瞄(Backtracking),乃是由於採用最左邊推演(Leftmost Derivation) 時,常會因選擇推演規則錯誤而必須回頭. 尤其是一具有Left Recursive 的文法,常使得在parsing 時會進入一個無窮循環 (Infinite Loop) 的情況,故必須事先將此一情況去除.

遞迴繼承構文解析程式(2/8)

2. 乃利用一些遞迴程序(Recursive Procedure)來辨認輸入的字串 且無需做回頭掃瞄的動作. 因為遞迴繼承構文解析程式是藉由遞迴程序來作剖析 因而對於具有Left Recursive 或Left Factor的文法,常使得在parsing 時會進入一個無窮循環(Infinite Loop) 的情況.故必須事先將此一情況去除.

遞迴繼承構文解析程式(3/8)

①Left recursive去除的方法

若有文法規則為

$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m$

則改為

$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_m A'$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_n A' | \varepsilon$

遞迴繼承構文解析程式(4/8)

② Left Factoring去除的方法

若有文法規則為

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_m | \gamma$

則改為

$A \rightarrow \alpha A' | \gamma$

$A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_m$

遞迴繼承構文解析程式(5/8)

- 範例:(Left Recursive)

G: $E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E)|id$

=>

G': $E \rightarrow TE'$

$E' \rightarrow +TE'|\varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT'|\varepsilon$

$F \rightarrow (E)|id$

遞迴繼承構文解析程式(6/8)

- 範例:(Left Factor)

G: $S \rightarrow iEtS|iEtSeS|a$

$E \rightarrow b$

=>

G': $S \rightarrow iEtSS'|a$

$S' \rightarrow eS|\varepsilon$

$E \rightarrow b$

遞迴繼承構文解析程式(7/8)

③ 若經過兩次以上的推演後才發生Left Recursive的情況,則稱之為間接左遞迴.

從文法的第一條規則依次往下處理,若目前正在處理文法的第I條規則,則將前面的I-1條規則分別代入第I條規則中.如有左遞迴之情況,再用上述消除左遞迴的方法予以消除

遞迴繼承構文解析程式(8/8)

- 範例:(Left Recursive)

  G: $S \to Aa|b$

  $A \to Ac|Sd|e$

將 $S \to Aa|b$ 代入 $A \to Ac|Sd|e$ 中

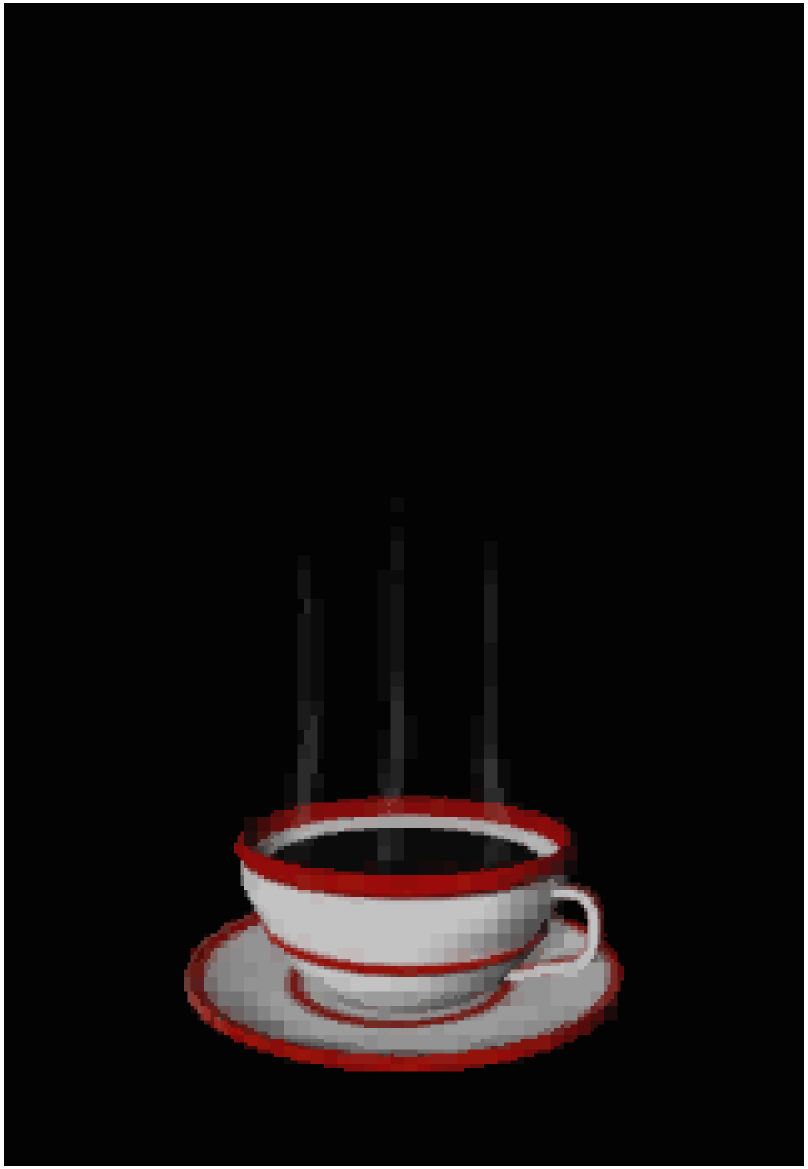
得 $A \to Ac|Aad|bd|e$,其中 $A \to Ac|Aad$ 為左遞迴

故 G’: $S \to Aa|b$

$A \to bdA'|eA'$

$A' \to cA'|adA'|\varepsilon$

- **Break time**



預期性的構文解析程式

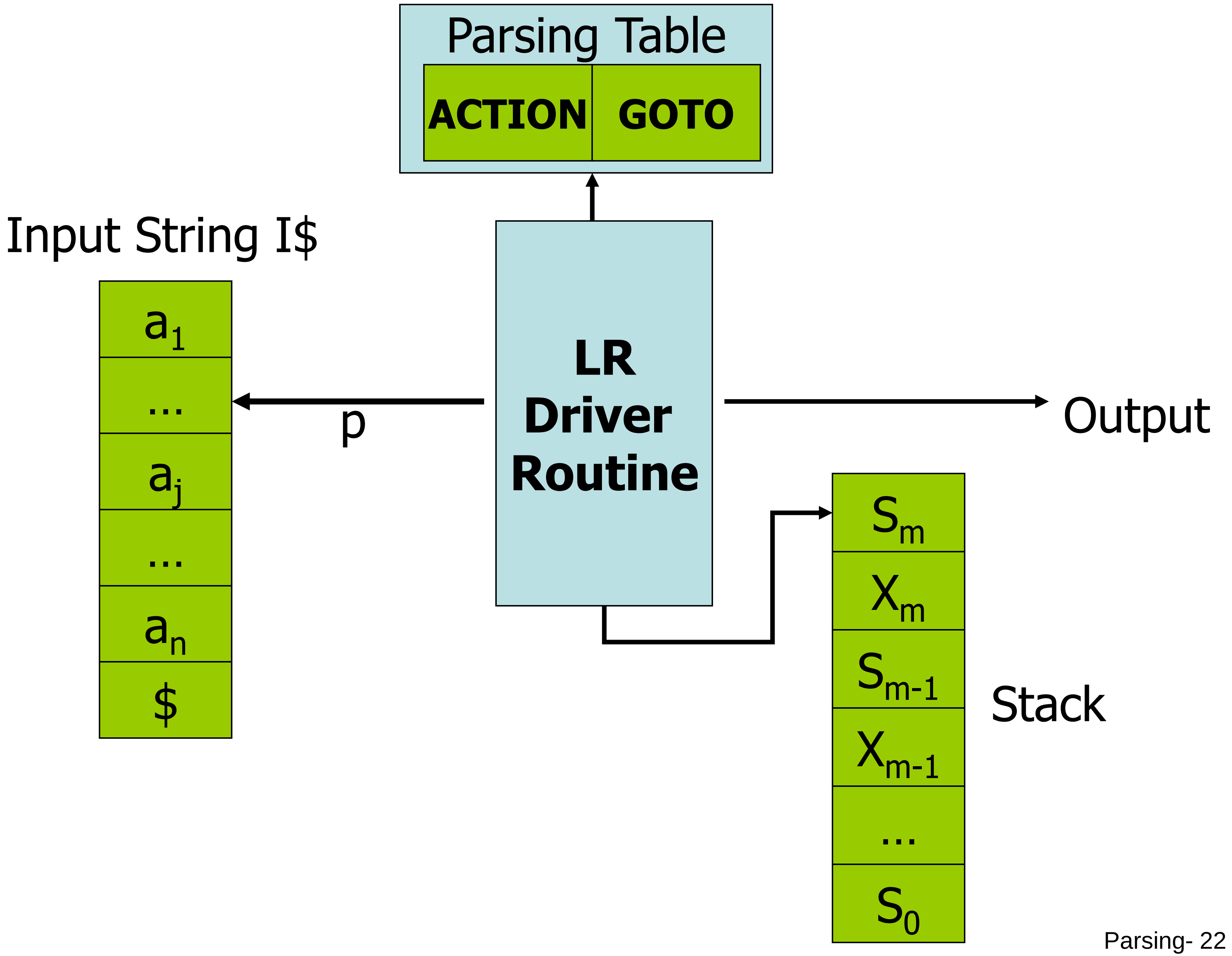
- Predictive Parser
  - 藉由一表格(Table) 來免除遞迴呼叫。
- Parser table
  - M[A,a] 二維陣列
  - A : 非終端符號
  - a :終端符號或終止符號$
  - 開始時,堆疊底部包含有終止符號$及開始符號

預期性的構文解析程式

- (See Slides)

LR Parser (Left To Right) Parser

- Contains 2 parts
  - Driver Routine
  - Parsing Table
    - ACTION
    - GOTO
- Processing Method
  - Left to Right Scan
  - Leftmost Reduction


Parsing Table
ACTION
GOTO
Input String I$
a_1
…
a_j
…
a_n
$
p
LR Driver Routine
Output
S_m
X_m
S_{m-1}
X_{m-1}
…
S_0
Stack

- String In Stack, $S_0X_1S_1X_2S_2....X_mS_m$
  - X_m:Symbol in Grammar Rule
  - S_m State
- LR Drive Routine
  - The state ,S_m in the Top of Stack
  - currently input symbol a_i
  - Parsing Table ACTION[S_m ,a_i]
  - 4 ACTIONS
    - Shift S
    - Reduce A->B
    - Accept
    - Error
  - The content of GOTO[S_m ,a_i] is the next state produced by S_m ,and a_i

- **LR Parser**
  - 藉由Stack的內容與未處理的輸入符號來表示LR Parser 的狀態,稱之為Configuration 如 $(S_0X_1S_1X_2S_2\ldots X_mS_m , a_ia_{i+1}a_{i+2}\ldots a_n\$)$
- **LR Drive Routine Actions**

(1)If ACTION $[S_m,a_i]$=Shift S, then

A new Configure=$(S_0X_1S_1X_2S_2\ldots X_mS_ma_iS , a_{i+1}a_{i+2}\ldots a_n\$)$

Symbol a_i is shifted into Stack

State S is push on the top of Stack

Currently input symbol is Symbol a_{i+1}

Endif

(2)If ACTION$[S_m,a_i]$=Reduce A->B, then

If StringLength(B)=k then

Pop 2k Symbols from Stack and then

the current state on the top of stack is S_{m-k}.

Push A on the top of Stack

By following GOTO$[S_{m-k},a_i]$ =S ,push S into the top of Stack

Currently input symbol is still Symbol a_i .

Current Configure=$(S_0X_1S_1X_2S_2\ldots X_{m-k}S_{m-k}AS , ,a_ia_{i+1}a_{i+2}\ldots a_n\$)$

Endif

LR Drive Routine Actions

(3)If ACTION [S_m,a_i]=Accept, then

Parsing is finished.

Endif

(4)If ACTION [S_m,a_i]=Error, then

The input string syntax error, and

Call error() routine.

Endif

LR Parsing Algorithm

```
While (true) do
Begin
    S is the state on the top of Stack and
    a is the symbol pointed by p;
    IF (ACTION[S,a]=Shift S') then begin
            push a and S' orderly on the top of Stack
            p points to next input symbol.
            End
    else    IF (ACTION[S,a]=Reduce A->B) then begin
                        pop 2k symbols; /* length(B)=k */
                        IF (current state in the top of stack is S')
                        Push A into the top of Stack.
                        Push the state of GOTO[S',A] on the top of Stack
                        end
    else     if (ACTION[S,a]=ACCEPT) then
                        return;
            else ERROR();
End
```

Example

G:
(1) S->S-T
(2) S->T
(3) T->T*F
(4) T->F
(5) F->[S]
(6) F->id

State	ACTION						GOTO		
	id	-	*	[	]	$	S	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			s11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

SLR Parsing Table

Input id*id-id

Step	Stack	Input Buffer	Output	
1	0	id*id-id$	Shift	
2	0id5	*id-id$	Reduce	F->id
3	0F3	*id-id$	Reduce	T->F
4	0T2	*id-id$	Shift	
5	0T2*7	id-id$	Shift	
6	0T2*7id5	-id$	Reduce	F->id
7	0T2*7F10	-id$	Reduce	T->T*F
8	0T2	-id$	Reduce	S->T
9	0S1	-id$	Shift	
10	0S1+6	Id$	Shift	
11	0S1+6id5	$	Reduce	F->id
12	0S1+6F3	$	Reduce	T->F
13	0S1+6T9	$	Reduce	S->S-F
14	0S1	$	Accept	

LR Parser 的優點

- 可用來辨認以Context-Free Grammar所描述的Programming Language.
- 無需回頭掃描(Backtracking), Better Performance and more popular Parser.
- 由Left-to-Right 來掃描輸入字串,能儘快地偵測出Syntactic Errors.

LR Parser

- 3 LR Parsers
  - SLR (Simple LR Parser)
  - LR(1) (Canonical LR Parser)
  - LALR (Look-ahead LR Parser)
- 若一文法G 被LR Parser 剖析時, 為了要決定Shift or Reduce的動作, 必須在檢查至當時輸入符號後的K個符號才能決定其動作, 則稱此文法為LR(K)文法.

SLR Parsing Table

(1)Grammar G 的一個LR(0) item 或稱 item.

Example

Grammar X->ABC

4 LR(0) items

X->•ABC

X->A • BC

X->AB • C

X->ABC •

(2)LR(0) item 所組合的集合稱之為 Canonical LR(0) Collection.此為製作LR Parser的基礎.

SLR Parsing Table

(3)擴大的文法(Augmented Grammar)

若文法G是以一個符號S為開始符號的文法當加入一個新符號S'及文法S'->S, 則形成文法G的擴大文法G'.

(4)Closure Operation

若I是文法G的一個項目集合, 則CLOSURE(I)之item集合可依據下列規則建立

Rule 1:每一個於I集合中的item, 皆是CLOSURE(I)中的item.

Rule 2:若A->$\alpha \bullet B\beta$ 是CLOSURE(I)集合中的item, 而且存在一文法規則 B->γ, 則將B->$\bullet\gamma$ 加入CLOSURE(I)中直到無法在加入新的item為止.

```
Procedure CLOSURE(I)
Begin
    Y:=I;
    Repeat
        For (於Y集合中存在一item A->α•Bβ
                且 有一文法規則B->γ
                且 B->•γ 不存在於Y集合中）do
                將B->•γ 加入Y集合中;
    Until (無其他item可在加入Y集合中);
    Return Y;
End;
```

Example

Augmented Grammar G'

G':

S'->S

S->S-T|T

T->T*F|F

F->[S]|id

若item I 為 {S'->S},則CLOSURE(I) 的所有item 集合為

S'->•S

S->•S-T

S->•T

T->•T*F

T->•F

F->•[S]

F->•id

- Item可分為兩類
  - Kernel item:包括起始item S′->S 以及所有“•”不出現在最左邊的item.
  - Non-kernel item:指“•”出現在最左邊的item
- GOTO[I,X]
  - I : an item集合
  - X : a Grammar Symbol
- GOTO[I,X]之意義
  - 若[A->α • Xβ]在I集合中, 則[A->α X • β]及其產生的CLOSURE皆在GOTO(I,X)中.

```
Procedure ITEM(G') /* Canonical LR(0) Collection D */
Begin
  D:={CLOSURE({S'->S});
  Repeat
      For (於D中的每一個I項目集合與文法符號X,
      使得GOTO(I,X)不是空集合且不存在於D中）do
            將GOTO(I,X)加入D中;
  Until (無其他item可在加入D集合中);
End;
```

Example :文法G’之Canonical LR(0) 項目集合的建構(1/4)

G’:S’->S

S->S-T|T

T->T*F|F

F->[S]|id

(1)I_0 : **{S’-> • S**

S-> • S-T

S-> • T

T-> • T*F

T-> • F

F-> •[S]

F-> • id}

1.GOTO(I_0,S)

I_1 : **{S’-> S •**

S-> S • -T}

2.GOTO(I_0,T)

I_2 : **{S’-> S •**

S-> S • -T}

3.GOTO(I_0,F)

I_3 : **{T-> F •}**

4.GOTO(I_0,[)

I_4 : **{F-> [• S]**

S-> • S-T

S-> • T

T-> • T*F

T-> • F

F-> • [S]

F-> • id}

5.GOTO(I_0,id)

I_5 : **{F-> id • }**

Example :文法G'之Canonical LR(0) 項目集合的建構(2/4)

(2)

1.GOTO(I_1,-)

I_6 :{S->S-•T

T->•T*F

T->•F

F->•[S]

F->•id}

(3)

1.GOTO(I_2,*)

I_7 :{T->T*•F

F->•[S]

F->•id}

(4)

1.GOTO(I_4,S)

I_8 :{ F->[•S]

S->S •−T}

2.GOTO(I_4,T)

I_2 :{S->T•

T->T•*F}

3.GOTO(I_4,F)

I_3 :{T->F•}

4.GOTO(I_4,[)

I_4 :{F->[•S] }

5.GOTO(I_4,id)

I_5 :{F->ID•}

Example :文法G'之Canonical LR(0) 項目集合的建構(3/4)

(5)

1.GOTO(I_6,T)

I_9 :{S->S-T •

T->T•*F}

(6)

1.GOTO(I_7,F)

I_{10} :{T->T*F •}

2.GOTO(I_7,[)

I_4 :{F->[•S] }

3. GOTO(I_7,id)

I_5 :{F->id• }

(7)

1.GOTO(I_8,])

I_{11} :{F->[S] •}

2.GOTO(I_8,-)

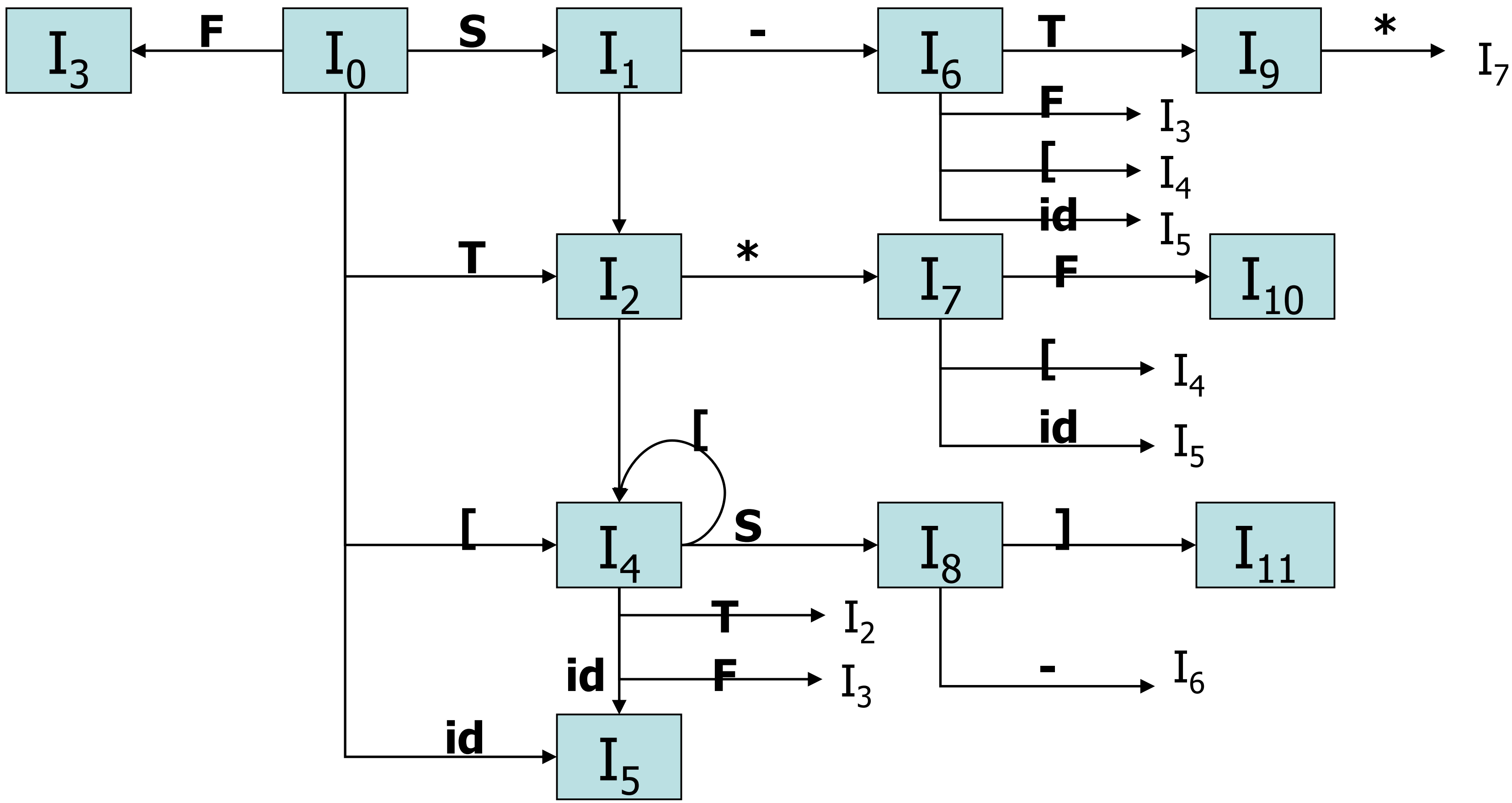
I_6 :{S->S-•T }

(8)

1. GOTO(I_9,*)

I_7 :{T->T*•F }

Example :文法G'之Canonical LR(0) 項目集合的建構(4/4)



Transition Diagram

SLR Parsing Table 建構的方法(1/2)

1. 建立Augment Grammar G' 的所有LR(0)的項目集合I_0 I_1 I_2 ,.... I_n
2. 每一個LR(0)項目集合是唯一新的State;即將I_i視為State i. 至於每個state 的動作則根據以下規則決定
   - 若[A-> α • aβ] 是在 I_i 中,而且GOTO(I_i ,a)=I_j, 則 ACTION[I_i,a]= Shift j. (a : Terminal Symbol)
   - 若[A-> B•] 是在 I_i 中,則令ACTION[I_i, a]="Reduce A->B".在此a 表示FOLLOW (A)中的元素.
   - 若[S'->S •]是在 I_i 中,則令ACTION[I_i, $]=Accept.

SLR Parsing Table 建構的方法(2/2)

3. 對於Non-terminal Symbol A, 若有GOTO(I_i ,A)= I_j, 則 GOTO[i,A]=j.
4. 其他未加以定義的欄位則表示錯誤(Error).
5. 令包含有[S'->S]的State 為起始狀態(Initial State).

範例:SLR Parsing Table 建構(1/3)

G: (1)S->S-T
(2)S->T
(3)T->T*F
(4)T->F
(5)F->[S]
(6)F->id

依前述方法=>
FOLLOW(S)={$,-,]}
FOLLOW(T)={$,-,],*]
FOLLOW(F)={$,-,],*}

1. **I_0:{S'-> • S**
   S-> •S-T
   S-> •T
   T-> •T*F
   T-> •F
   F-> •[S]
   F-> •id}

ACTION[0,[]=Shift 4
ACTION[0,id]=Shift 5
GOTO[0,S]=1
GOTO[0,T]=2
GOTO[0,F]=3

範例:SLR Parsing Table 建構(2/3)

2. I_1:{S′-> S •
   S-> S • -T}

ACTION[1,$]=Accept
ACTION[1,-]=Shift 6

3. I_2:{S-> T•
   T-> T• *F}

ACTION[2,*]=Shift 7

因為 FOLLOW(T)={$,-,],*},
∴
Action[2,$]="Reduce S->T"
Action[2,-] "Reduce S->T"
Action[2,]] "Reduce S->T"
Action[2,*] "Reduce S->T"

範例:SLR Parsing Table 建構(3/3)

依此類推….We will get SLR Parsing Table

State	ACTION						GOTO		
	id	-	*	[	]	$	S	T	F
0	S5			S4			1	2	3
1		S6				Acc			
2		R2	S7		R2	R2			
3			R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6				S11			
9		S7	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

- 如果在SLR Parsing Table中的某一欄位同時具有兩個Action時,則此文法非SLR(1)文法

Shift-Reduce Conflict

若Grammar

G: S->L+K

S->K

L->*K

L->id

K->L

可導出所有的LR(0)的項目 $\mathbf{I_0\ I_1\ I_2 \ldots\ I_n}$

其中 $\mathbf{I_2}$:{S->L • +K

K->L •}

產生Shift-Reduce Conflict.

由 S->L • +K =>ACTION[2,+]=Shift 6

K->L • =>ACTION[2,+]="Reduce K->L" (FOLLOW(K)={+,$})

故此文法非SLR(1)文法

SLR ,LALR,LR(1) 比較

Parser	**功能**	**成本**	**No of States**
SLR			
LALR			
LR(1)			