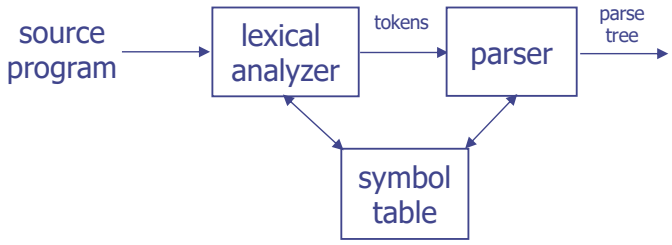
編譯器設計

Chapter 4
Syntax Analysis

Syntax Analysis

- The role of the syntax analyzer
  - To accept a string of tokens from the scanner
  - To verify the string can be generated by the grammar



- Three general types of parsers for grammars
  - Universal parsing methods
    - e.g. Cocke-Younger-Kasami, Earley's
  - Top-down
  - Bottom-up

too complicated 一般不用

從 root 建 $a = b + c \Rightarrow id = id + id$

從 bottom 建 $id = id + id \Rightarrow a = b + c$

都是建 tree

較常用

Syntax Error Handling

- The goals of the error handler in a parser
  - It should report the presence of errors clearly and accurately
  - It should recover from each error quickly enough to be able to detect subsequent errors
  - It should not significantly slow down the processing of correct programs
- Many errors could be classified [Ripley et al 1978]:
  - 60% punctuation errors
  - 20% operator and operand errors
  - 15% keyword errors
  - 5% others

Error Recovery Strategies

- Panic mode
  - The simplest method and can be used by most parsers
  - On discovering an error, the parser discards input symbols one at a time until one of synchronizing tokens is found
    - The synchronizing tokens are usually delimiters, e.g. ; or end
- Phrase level
  - On discovering an error, a parser may perform local correction on the remaining input
    - e.g. replacing a comma by a semicolon, inserting a missing ;
- Error production
  - Augment the grammar with productions that generate the erroneous constructs
- Global correction
  - Make as few changes as possible in processing the input

Context-Free Grammars

A → α

<stmt> → id = <expr>

- The syntax of programming language constructs can be described by *context-free grammars*
  - Or BNF (Backus-Naur Form)
- Grammars offer significant advantages
  - A grammar gives a precise syntactic specification of a programming language
  - From certain classes of grammars we can automatically construct an efficient parser
    - The parser construction process can reveal syntactic ambiguities and difficult-to-parse constructs 因為產不出tree
  - A properly designed grammar facilitates the translation into target code
  - New language constructs can be added easily

Context-Free Grammars

BNF描述的就是 Context-Free Grammars

- A context-free grammar G can be denoted by G = (V_N, V_T, P, S)
  - V_N: nonterminals
    - Syntactic variables that denotes sets of strings
  - V_T: terminals
    - The basic symbols (i.e. tokens) from which strings are formed
  - P: productions
    - The manner in which the terminals and nonterminals can be combined for form strings, e.g.

      expr → *expr op expr*

      expr → id

      op → + | - | * | /
  - S: start symbol

Why Not Using Regular Languages

表達能力不足

- Regular languages are not power enough to represent certain constructs in program languages, e.g. $\{^n\}^n$ usually called "balanced"
  - i.e. $\{^n\}^n$ is not a regular language

regular language : $\{^*\}^*$ 但不能表達"幾個"無法表達對稱

- Recall the ways to prove a language is regular
  - Find a regular grammar (or regular expressions) that generates the language
  - Find an NFA that recognizes the language

如要證明他"是" RL

- However, they can not prove $\{^n\}^n$ is not regular
  - The pumping lemma of regular languages will be used
  - Prove by contradiction

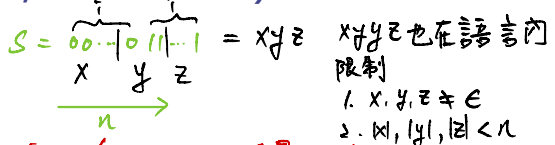
要用 Context-Free Language

只要是 Regular Language 就有此特性

Pumping Lemma

語言：句子的集合

- If A is a regular language and string $s \in A$, where $\exists n \in N$ such that $|s| \geq n$, then $s = xyz$ where
  - $xy^iz \in A$, $i \geq 0$,
  - $|y| > 0$, and
  - $|xy| \leq n$

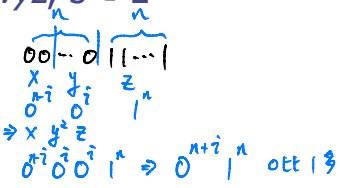


證明不是：pumping lemma ⇒ 不是 regular

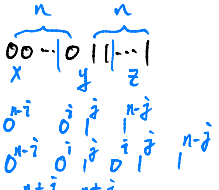
- Example: prove $L = \{0^n1^n \mid n \geq 0\}$ is not regular
  - Assume L is regular. Let $s = wyz$, $s \in L$
  - If y consists of 0's only, then xy^iz has more 0's than 1's
  - If y consists of 1's only, then xy^iz has more 1's than 0's
  - If y consists of 0's and 1's, then $xy^iz \neq 0^n1^n$

超過上限 n ⇒ 代表中間一定有東西 ⇒ y

類似生male原理



G = {{S}, {0,1}}
P = { S → 01
S → 0S1 }
0^n1^n



$L = 0^p 1^{p+1}$ False

xyz: $x = 0^\alpha$

$y = 0^\beta \quad \beta > 0$

$z = 0^{p-\alpha-\beta} 1^{p+1}$

Look at xy^iz

$xy^iz = 0^\alpha 0^{i\beta} 0^{p-\alpha-\beta} 1^{p+1}$

$= 0^{p+i\beta-\beta} 1^{p+1}$

$\Rightarrow p + i\beta - \beta < p + 1$

$\Rightarrow \beta(i-1) < 1$

we know $\beta > 0 \rightarrow \beta \geq 1$

$L = \{ 0^{2^n} 1^n : n \geq 0 \}$ False

Suppose L is regular

$\Rightarrow$ exist a p for L

Choose $w = 0^{2^p} 1^p$

Look at all decomp of w into

xyz:

$x = 0^\alpha$

$y = 0^\beta, \ \beta \geq 1$

$z = 0^{2^p - \alpha - \beta} 1^p$

Choose i s.t. $xy^iz \notin L$

xy^iz:

$0^\alpha 0^{i\beta} 0^{2^p-\alpha-\beta} 1^p$

$= 0^{2^p + \beta(i-1) - \alpha} 1^p$

$2^p + \beta(i-1) = 2^p$

$\beta(i-1) = 0$

$\beta \geq 1, \ i = 1$

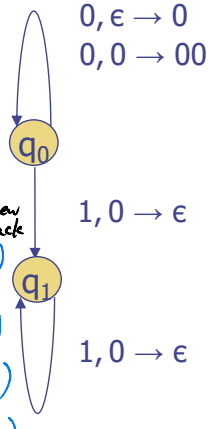
if $i = 2$, it's not regular language any more.

辨認 context-free language

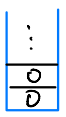
Pushdown Automata 多 stack NFA + Stack

- $M = (K, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ 固定 pop stack 的頂端
  - K : Finite set of states
  - Σ : Input alphabet
  - Γ : Pushdown alphabet (gamma)
  - δ : Mapping $K \times (\Sigma \cup \epsilon) \times \Gamma \rightarrow K \times \Gamma^*$ (delta) — stack (gamma closure)
  - q_0 : Initial state ($q_0 \in K$)
  - Z_0 : Initial pushdown symbol ($Z_0 \in \Gamma$)
  - F : Set of final states
- Example: $L = \{0^n 1^n \mid n \geq 0\}$
  - $M = (\{q_0, q_1\}, \{0, 1\}, \{0, 1\}, \delta, q_0, \epsilon, \emptyset)$
    - $\delta(q_0, 0, \epsilon) = (q_0, 0)$ ← top of stack (empty)
    - $\delta(q_0, 0, 0) = (q_0, 00)$ ← 拿出 top of stack
    - $\delta(q_0, 1, 0) = (q_1, \epsilon)$ 拿出去，最後放回來的
    - $\delta(q_1, 1, 0) = (q_1, \epsilon)$

現在還要看 stack 的資料



Regular Language $\{0^*1^*\}$



有 0 就放入 stack
看到 1 就拿出 0
可以達成 0^n1^n

1. final state
2. empty stack

有這兩種方法決定有沒有結束

NFA

$M = \{S, \Sigma, \delta, s_0, F\}$
- S: a finite, nonempty set of *states*
- Σ: an *alphabet*
- δ: mapping $S \times \Sigma \rightarrow 2^S$
- s_0: start state
- F: the set of accepting (or final) states

$\delta = (s_0, 0, \epsilon) = (s_0, n)$ (state, input, stack top → new stack)

$S \rightarrow 0S1$, 0^n1^n $n \geq 1$

$\delta(q_0, \epsilon, 0) \rightarrow (q_1, 0)$

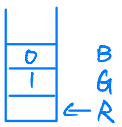
$\delta(q_1, 1, 0) \rightarrow (q_1, \epsilon)$

0 已經 pop 出來了

Pushdown Automata

- Example: $L = \{w c w^R \mid w \in \{0, 1\}^*\}$ (reverse)
  - $M = (\{q_0, q_1\}, \{0, 1, c\}, \{R, G, B\}, \delta, q_0, R, \emptyset)$
    - $\delta(q_0, 0, R) = (q_0, BR)$ Stack 裡面一開始不是空的
    - $\delta(q_0, 0, G) = (q_0, BG)$
    - $\delta(q_0, 0, B) = (q_0, BB)$
    - $\delta(q_0, 1, R) = (q_0, GR)$
    - $\delta(q_0, 1, G) = (q_0, GG)$
    - $\delta(q_0, 1, B) = (q_0, GB)$
    - $\delta(q_0, c, R) = (q_1, R)$
    - $\delta(q_0, c, G) = (q_1, G)$
    - $\delta(q_0, c, B) = (q_1, B)$
    - $\delta(q_1, 0, B) = (q_1, \epsilon)$
    - $\delta(q_1, 1, G) = (q_1, \epsilon)$
    - $\delta(q_1, \epsilon, R) = (q_1, \epsilon)$

pop 完 R, stack = empty



0011c1100 / 1010c0101

B=0, G=1

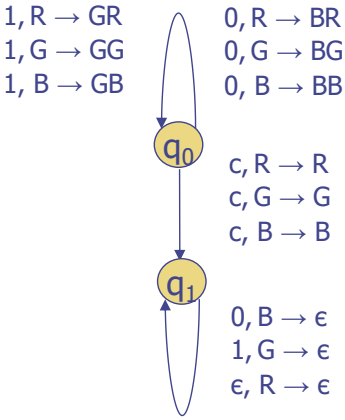
$\delta(q_0, 0, R) = (q_0, BR)$
$\delta(q_0, 1, R) = (q_0, GR)$

$\delta(q_0, 0, B) = (q_0, BB)$
$\delta(q_0, 0, G) = (q_0, GB)$
$\delta(q_0, 1, B) = (q_0, BG)$
$\delta(q_0, 1, G) = (q_0, GG)$

$\delta(q_0, c, G) = (q_1, G)$
$\delta(q_0, c, B) = (q_1, B)$
$\delta(q_0, c, R) = (q_1, R)$

$\delta(q_1, 1, G) = (q_1, \epsilon)$
$\delta(q_1, 0, B) = (q_1, \epsilon)$

Reverse 後一樣，pop 出最上面，return ϵ



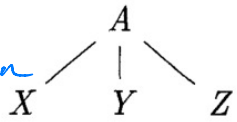
$\delta(q_1, 0, B) = (q_1, \epsilon)$

input / top of stack 會拿出來

放入 stack 的內容有些會直接拿掉，有些會拿掉再放回去，甚至加新的東西

Parse Trees

- A parse tree pictorially shows how the start symbol of a grammar derives a string in the language
  - e.g. A → XYZ

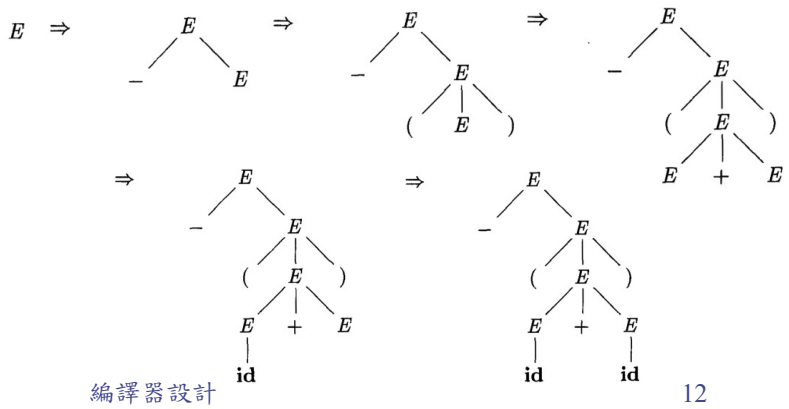


- Formally, given a context-free grammar, a parse tree is a tree with the following properties
  - The root is labeled by the start symbol
  - Each leaf is labeled by a token or by ϵ
  - Each interior node is labeled by a nonterminal
  - If A is the nonterminal labeling some interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then $A \rightarrow X_1 X_2 \ldots X_n$ is a production

Parse Trees and Derivations

- There are several ways to view the process by which a grammar defines a language
  - Building parse trees
  - Derivations
- Derivations give a precise description of the top-down construction of a parse tree
  - Example: $E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \text{id}$
  - w = - (id + id)
  - Derivation:

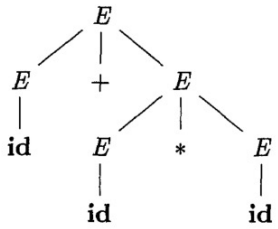
$E \Rightarrow -E$
$\Rightarrow -(E)$
$\Rightarrow -(E + E)$
$\Rightarrow -(\text{id} + E)$
$\Rightarrow -(\text{id} + \text{id})$



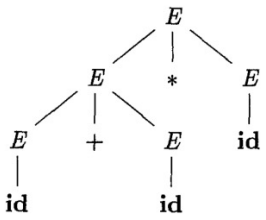
Parse Trees and Derivations

- Every parse tree has associated with it a unique leftmost and a unique rightmost derivation
  - However, not every sentence has exactly one leftmost or rightmost derivation
  - e.g. id + id * id

$E \Rightarrow E + E$
$\Rightarrow$ id + E
$\Rightarrow$ id + $E * E$
$\Rightarrow$ id + id * E
$\Rightarrow$ id + id * id



$E \Rightarrow E * E$
$\Rightarrow E + E * E$
$\Rightarrow$ id + $E * E$
$\Rightarrow$ id + id * E
$\Rightarrow$ id + id * id

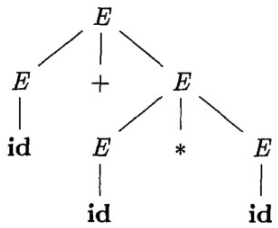


Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*
  - i.e. more than one leftmost or rightmost derivation for the same sentence
- Two options to deal with ambiguity
  - *Disambiguating rules* are used to throw away undesirable parse trees
  - To eliminate ambiguity 跟左/右計算
    - Precedence and associativity
    - Rewriting the grammar

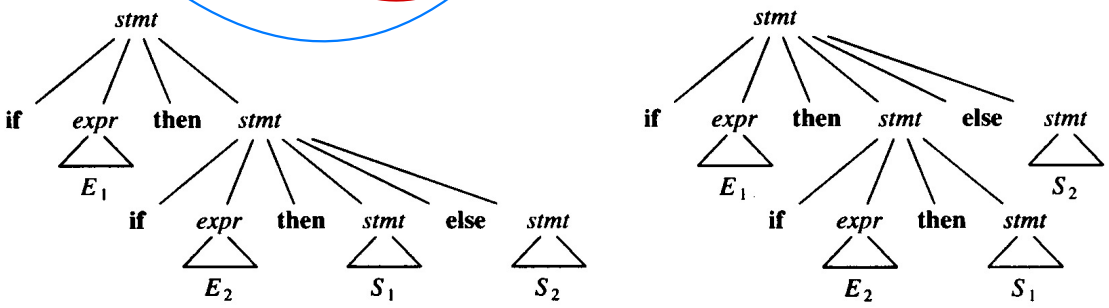
Example: $E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid$ id

Rewrite : $E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$ Term
$F \rightarrow (E) \mid$ id Factor (最低階)



Eliminating Ambiguity

- Example (Dangling Else):
  - (1) *stmt* → if *expr* then *stmt*
  - (2) *stmt* → if *expr* then *stmt* else *stmt*
  - *stmt* → other
  - if E_1 then (if E_2 then S_1) else S_2

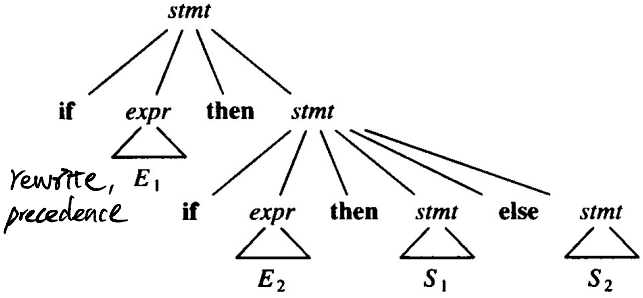


Eliminating Ambiguity

- Rewrite the grammar:

stmt → matched_stmt
| unmatched_stmt
matched_stmt → if *expr* then *matched_stmt* else *matched_stmt*
| other
unmatched_stmt → if *expr* then *stmt*
| if *expr* then *matched_stmt* else *unmatched_stmt*

Ambiguity
1. E+E | E*E
2. dangling else
3. left recursion

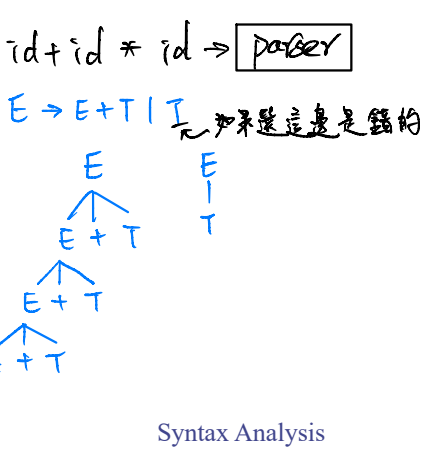


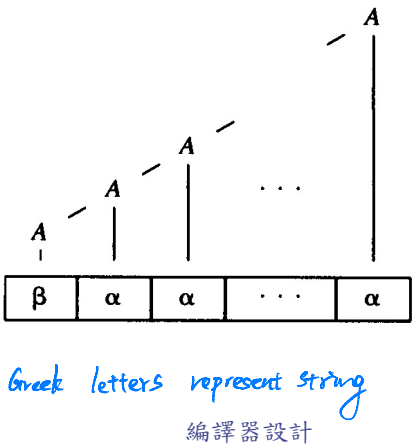
rewrite, precedence

multiple if, one else. Don't know which 'if' is matched.

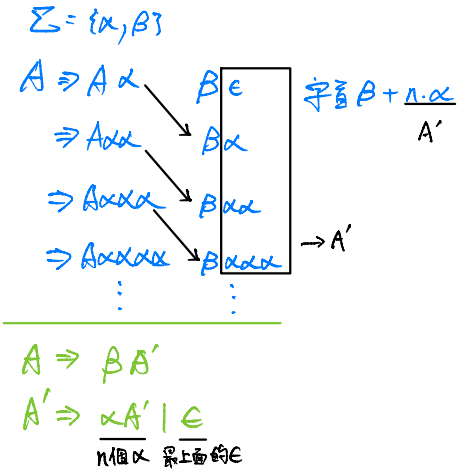
Left Recursion

- A grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$
  - e.g. $A \rightarrow A\alpha \mid \beta$ Compiler keeps choosing first derivation rule and causes Left Recursion.
- It is possible that a parser to loop forever if the grammar is left recursive





Greek letters represent string



Eliminating Left Recursion

- A left-recursive pair of production $A \rightarrow A\alpha \mid \beta$ can be changed to non-left-recursive productions

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' \mid \epsilon$

  without changing the set of strings derivable from A
- Example:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

$A \Rightarrow A\alpha \mid \beta$

$\Rightarrow A \Rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

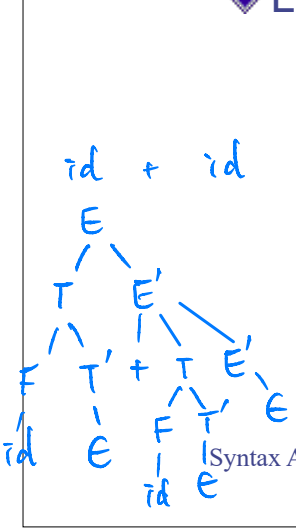
$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

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



id + id

$E \Rightarrow TE'$

$\Rightarrow FT'E'$

$\Rightarrow \text{id}\,T'E'$

$\Rightarrow \text{id}\,\epsilon E'$

$\Rightarrow \text{id}\,\epsilon + TE'$

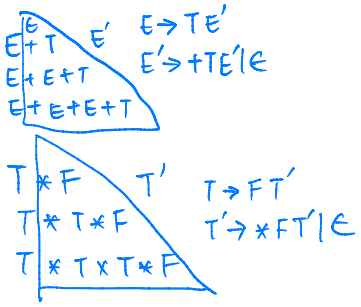
$\Rightarrow \text{id}\,\epsilon + FT'E'$

$\Rightarrow \text{id}\,\epsilon + \text{id}\,T'E'$

$\Rightarrow \text{id}\,\epsilon + \text{id}\,\epsilon E'$

$\Rightarrow \text{id}\,\epsilon + \text{id}\,\epsilon\epsilon$

$\Rightarrow \text{id} + \text{id}$



Eliminating Left Recursion

- Algorithm
  - Input. Grammar G with no cycles on ϵ-productions
  - Output. An equivalent grammar with no left recursion
  - Method.

```
Arrange the nonterminals in some order A1, A2, ..., An
for i = 1 to n do
  for j = 1 to i-1 do
    replace each production of the form Ai → Ajγ by
      the production Ai → δ1γ | δ2γ | ... | δkγ,
      where Aj → δ1 | δ2 | ... | δk are all the current Aj-productions
    eliminate the immediate left recursion among the Ai-productions
  end
end
```


Eliminating Left Recursion

- Example

  $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Sd \mid \epsilon$

  - Arrange the nonterminals in order S, A
  - i = 1
    - No immediate left recursion among the S-productions
  - i = 2
    - Substitute the S-productions in $A \rightarrow Sd$ to obtain

      $A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$

    - Eliminate the left recursion and yield

      $S \rightarrow Aa \mid b$

      $A \rightarrow bdA' \mid A'$

      $A' \rightarrow cA' \mid adA' \mid \epsilon$

Midterm Range

Left Factoring

- Left factoring is a grammar transformation
  - Usually applied when it is not clear which of two alternative productions to use to expand a nonterminal *A*
  - Rewrite the *A*-productions to defer the decision
  - e.g. *stmt* → if *expr* then *stmt* else *stmt*
    stmt → if *expr* then *stmt*) 相同
- Algorithm
  - For each nonterminal *A* find the longest prefix α common to two or more of its alternatives
  - Replace $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ by
    $A \rightarrow \alpha A' \mid \gamma$
    $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

等到出現 else 再選

if E then S S'
S' ⇒ ∈ | else S
α β1 β2

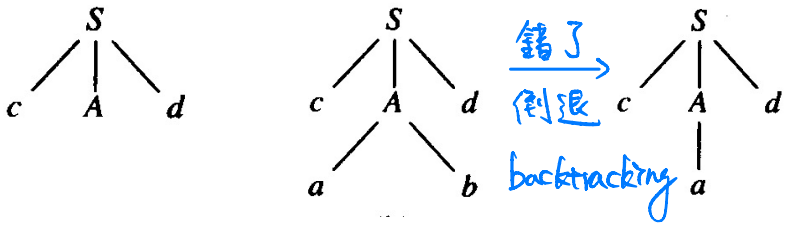
開頭是 terminal，後面是 nonterminal

$A \Rightarrow \alpha\beta_1 \mid \alpha\beta_2$
$A \Rightarrow \alpha A'$
$A' \Rightarrow \beta_1 \mid \beta_2$

Top-Down Parsing

$L = \{cad, cabd\}$

- Top-down parsing an be viewed as an attempt to find a leftmost derivation for an input string
  - Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder
- Example
  $S \rightarrow cAd$
  $A \rightarrow ab \mid a$



Top-Down Parsing

- Recursive-Descent Parsing
  - Start with the root, labeled with the starting symbol, and repeatedly perform the following steps
    - At node *n* labeled with nonterminal *A*, select one of the productions for *A* and construct children at *n* for the symbols on the right side of the production
    - When a terminal is added that does not match the input string, then backtrack
    - find the next node
- Predictive Parsing 不會猜錯的 parser
  - Recursive-descent parsing without backtracking

Designing a Predictive Parser

$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$

- A *predictive parser* is a program consisting a procedure for every nonterminal. Each procedure does two things

id + id → A (lookahead)

  - For all *A*-productions it decides which production to use by looking at the lookahead symbol, say a, i.e.

```
if (lookahead == "+") {
    match("+")
    T(); E();
```

指標往下指 match完呼叫自己

    - The production $A \to \alpha$ is used if
      $a \in \text{FIRST}(\alpha)$ 第一個字母的集合
      where FIRST(α) is the set of terminals that begins the strings derived from α, or
    - The production $A \to \epsilon$ is used if 選不到就 ϵ
      $a \notin \text{FIRST}(\alpha)$
  - The procedure uses a production by mimicking the right side:
    - A nonterminal results a call to the procedure for the nonterminal
    - A token matching the lookahead symbol results in the next input toke being read

遇到 terminal
→ match, point to next

Designing a Predictive Parser

- Example

  type → *simple* | ↑id | array [*simple*] of type

  simple → integer | char | num dotdot num

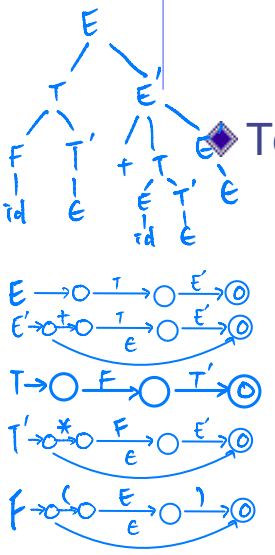
```
procedure match(t : token);
begin
  if lookahead = t then
    lookahead := nextoken
  else error
end;
```

```
procedure type;
begin
  if lookahead ∈ {integer, char, num} then
    simple
  else if lookahead = '^' then begin
    match('^'); match(id)
  end
  else if lookahead = array then begin
    match(array); match('['); simple;
    match(']'); match(of); type
  end
  else error
end;
```

```
procedure simple;
begin
  if lookahead = integer then
    match(integer)
  else if lookahead = char then
    match(char)
  else if lookahead = num then begin
    match(num); match(dotdot); match(num)
  end
  else error
end;
```


Transition Diagrams for Predictive Parsers

- Features
  - There is one diagram (procedure) for each nonterminal
  - The labels of edges are tokens (terminals) and nonterminals
  - A transition on a token is taken if that token is the next input symbol
  - A transition on a nonterminal *A* is a call of the procedure of *A*
- To construct the transition diagram from a grammar
  - Eliminate left recursion from the grammar
  - Left factor the grammar
  - For each nonterminal *A*, do the following
    - Create an initial state and a final (return) state
    - For each production $A \rightarrow X_1X_2...X_n$, create a path from the initial to the final state, with edges labeled X_1, X_2, ..., X_n



Transition Diagrams for Predictive Parsers

- Example

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \epsilon$

  $T \rightarrow FT'$

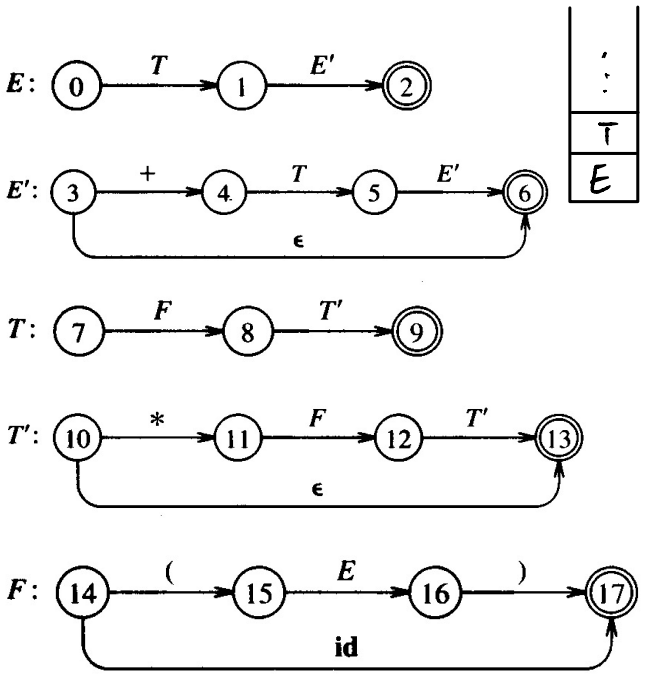
  $T' \rightarrow *FT' \mid \epsilon$

  $F \rightarrow (E) \mid id$

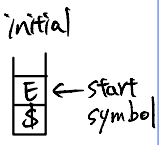
  - w = id + id * id

```
E( ) {
  T( ); E'( );
}

E'( ) {
  if (lookahead == '+') {
    match('+'); T( ); E'( );
  }
}
```

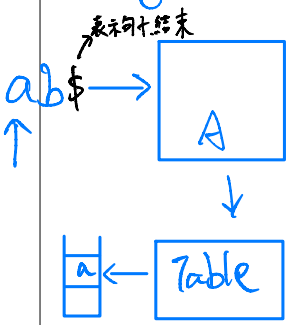


Nonrecursive Predictive Parser



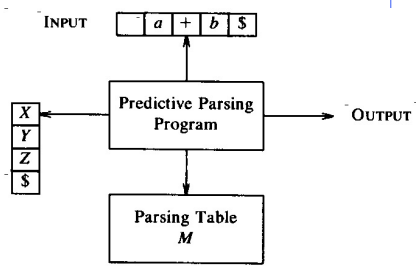
- The key problem during predictive parsing is that of determining the production to be applied for a nonterminal
- A table-drive predictive parser has an input buffer, a stack, a parsing table, and an output stream
- The action of the parser is determined by X, the symbol on top of the stack, and a, the current input symbol
  - If X = a = '$', the parser halts and announces successful completion of parsing
  - If X = a ≠ '$', the parser pops X off the stack and advances the input pointer to the next input symbol
  - If X is a nonterminal, the program consults entry M[X, a] of the parsing table M.
    - If, for example, M[X, a] = {$X \rightarrow UVW$}, the parser replaces X on top of the stack by WVU (with U on top).
    - If M[X, a] = error, the parser calls an error recovery routine.

產生不同的 parsing table.



match ⇒ pop stack 頂端, point to next

Nonrecursive Predictive Parser



- Algorithm
  - Input. A string *w*$ and a parsing table M
  - Output. A leftmost derivation of *w*
  - Method. Initially, stack = $S

```
set ip to point to the first symbol of w$
repeat
  let X be the top stack symbol and a the symbol pointed by ip
  if X is a terminal or $ then
    if X = a then
      pop X off the stack and advance ip
  else                    /* X is a nonterminal */
    if M[X, a] = X → Y1 Y2 ... Yk then
      pop X from the stack
      push Yk, Yk-1, ..., Y1 onto the stack, with Y1 on the top
      Output the production X → Y1 Y2 ... Yk
    else error( )
until X = $               /* stack is empty */
```


Nonrecursive Predictive Parser

Advantages
① only thing need to change is parsing table
② no recursion

- Example
  - $E \rightarrow TE'$
  - $E' \rightarrow +TE' \mid \epsilon$
  - $T \rightarrow FT'$
  - $T' \rightarrow *FT' \mid \epsilon$
  - $F \rightarrow (E) \mid \text{id}$
- Parsing Table
  - Fig. 4.17

空格是error → syntax error

NONTERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

看到 stack 頂端 → 查表格
→ 放回表格內的放到 top of stack
→ check if top matches Input point → if match → pop out, pointer move to next

Stack	Input	Output	Leftmost Derivation
$E	id + id * id$		$E \Rightarrow$
$E'T	id + id * id$	$E \rightarrow TE'$	$TE' \Rightarrow$
$E'T'F	id + id * id$	$T \rightarrow FT'$	$FT'E' \Rightarrow$
$E'T'\text{id}	id + id * id$	$F \rightarrow \text{id}$	$\text{id}\,T'E' \Rightarrow$
$E'T'	+ id * id$		
$E'	+ id * id$	$T' \rightarrow \epsilon$	$\text{id}E' \Rightarrow$
$E'T+	+ id * id$	$E' \rightarrow +TE'$	$\text{id} + T\,E' \Rightarrow$
$E'T	id * id$		
$E'T'F	id * id$	$T \rightarrow FT'$	$\text{id} + FT'E' \Rightarrow$
$E'T'\text{id}	id * id$	$F \rightarrow \text{id}$	$\text{id} + \text{id}\,T'E' \Rightarrow$
$E'T'	* id$		
$E'T'F*	* id$	$T' \rightarrow *FT'$	$\text{id} + \text{id} *FT'\,E' \Rightarrow$
$E'T'F	id$		
$E'T'\text{id}	id$	$F \rightarrow \text{id}$	$\text{id} + \text{id} * \text{id}\ T'E' \Rightarrow$
$E'T'	$		
$E'	$	$T' \rightarrow \epsilon$	$\text{id} + \text{id} * \text{id}\ E' \Rightarrow$
$	$	$E' \rightarrow \epsilon$	$\text{id} + \text{id} * \text{id}$

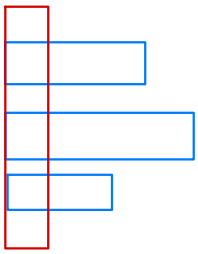
FIRST and FOLLOW

$A \to \alpha \Rightarrow w$

$B \to \beta \Rightarrow \gamma$

- The construction of a predictive parser is aided by two functions associated with a grammar *G*
- FIRST(α)
  - The set of terminals that begins the strings derived from α
  - To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added
    - If X is a terminal, then FIRST(X) = {X}
    - If $X \to \epsilon$ is a production, then add ϵ to FIRST(X)
    - If $X \to Y_1 Y_2 \ldots Y_k$ is a production, then
      - Place a in FIRST(X) if $\exists i\ (1 \le i \le k)$ a $\in$ FIRST(Y_i) and $\epsilon \in$ FIRST(Y_j) $\forall j\ (1 \le i < j)$
      - Place ϵ in FIRST(X) if $\forall i\ (1 \le i \le k)\ \epsilon \in$ FIRST(Y_i)

First(ϵ) = {ϵ}



收集句子的開頭做為 set 給 first.

First(X) = First(Y_1)

前面幾個 Y 會變空字串，就要把 $Y_1 Y_2 \ldots Y_n Y_{n+1}$ 都加入 First

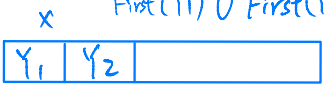
如果全部都會 First 要加入(ϵ)

① if $Y_1 \neq \epsilon$

② if $Y_1 = \epsilon$ (跳) 加入 Y_2

then First(Y_1) $\cup$ First(Y_2)

First(Y_1) $\cup$ First(Y_2) $\cup \ldots$ First(Y_{n-1}) $Y_1 \ldots Y_{n-1} \Rightarrow \epsilon$

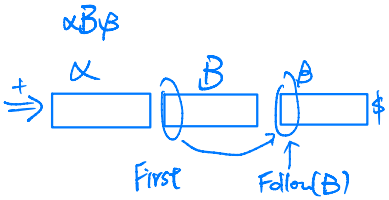


FIRST and FOLLOW

希臘字 → 字串

lowercase → terminal

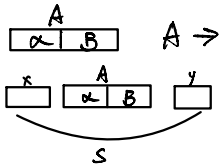
uppercase → non-terminal.



- FOLLOW(A) First 後面的第一個字母
  - The set of terminals a that can appear immediately to the right of A in some sentential form, i.e.
    a $\in$ FOLLOW(A) if there exists a derivation $S \Rightarrow \alpha A a \beta$
  - To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added
    1. Place \$ in FOLLOW(S)
    2. If there is production $A \to \alpha B \beta$, then add FIRST(β) - {ϵ} to FOLLOW(B)
    3. If there is production $A \to \alpha B$, then add FOLLOW(A) to FOLLOW(B)
    4. If there is production $A \to \alpha B \beta$ and $\epsilon \in$ FIRST(β), then add FOLLOW(A) to FOLLOW(B)

1. $A \to \alpha B \beta$ FIRST(β)

2. $A \to \alpha B$



$S \overset{*}{\Rightarrow} xAy$ (A 的 follow 也到 B 的 follow)

同 2.

FOLLOW 沒有 ϵ

FIRST and FOLLOW

- Example

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \epsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \epsilon$

  $F \rightarrow (E) \mid id$

- FIRST
  - FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
  - FIRST(E') = {+, ϵ}
  - FIRST(T') = {*, ϵ}
- FOLLOW
  - FOLLOW(E) = FOLLOW(E') = {), $}
  - FOLLOW(T) = FOLLOW(T') = {+,), $}
  - FOLLOW(F) = = {+, *,), $}

$E \rightarrow TE'$ $\quad E' \rightarrow +T$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$ $\quad T' \rightarrow *F$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

Follow(E) = {$} ∪ {)}

Follow(E') = Follow(E) = {$,)}

Follow(T) = First(E') ∪ Follow(E') = {+, $,)}

Follow(T') = Follow(T) = {+, $,)}

Follow(F) = First(T') ∪ Follow(T') = {*, +, $,)}

Constructing Predictive Parsing Table

- Algorithm
  - Input. Grammar *G*
  - Output. Parsing table M
  - Method.
    1. For each production $A \rightarrow \alpha$, do steps 2 to 4
    2. For each terminal a ∈ FIRST(α), then add $A \rightarrow \alpha$ to M[*A*, a]
    3. If ϵ ∈ FIRST(α), then add $A \rightarrow \alpha$ to M[*A*, b] for each terminal b ∈ FOLLOW(*A*)
    4. If ϵ ∈ FIRST(α) and $ ∈ FOLLOW(*A*), then add $A \rightarrow \alpha$ to M[*A*, $]
    5. Make each undefined entry of M be *error*

$X \rightarrow \alpha \mid \beta \mid \epsilon$

	a	b	c
X	X→α	X→β	Follow(X)

如果a∈αβ產生

⇒ FIRST(α), FIRST(β)

如果是ϵ

⇒ FOLLOW(X)

Constructing Predictive Parsing Table

- Example

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \epsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \epsilon$

  $F \rightarrow (E) \mid id$

	FIRST	FOLLOW
E	(id	) $
E'	+ ϵ	) $
T	(id	+) $
T'	* ϵ	+) $
F	(id	* +) $

Nonter-minal	Input Symbol					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ id			$F \rightarrow (E)$		

Follow(E')

Constructing Predictive Parsing Table

- Example

  $S \rightarrow iEtSS' \mid a$

  $S' \rightarrow eS \mid \epsilon$

  $E \rightarrow b$

(if E then S)

left factoring 的 example

- FIRST(S) = {i, a}
- FIRST(S') = {e, ϵ}
- FIRST(E) = {b}
- FOLLOW(S) ={e, $}
- FOLLOW(S') = {e, $}
- FOLLOW(E) = {t}

	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S'→eS S'→ϵ			S'→ϵ
E		E→b				

Follow(S) = First(S') ∪ $ ∪ Follow(S') = {e, $}

Follow(S') = Follow(S) = {e, $}

Follow(E) = {t}

一格多個：ambiguous

LL(1) Grammars

predict parser 只能處理 LL(1)
top down parsing.

- A grammar whose predictive parsing table has no multiply-defined entries is said to be *LL*(1)
  - 1st *L*: scanning the input from left to right
  - 2nd *L*: producing a leftmost derivation
  - 1: using one input symbol for lookahead at each step
- Properties
  - Not ambiguous
  - Not left recursive

left recursion 只會出現在 Top-Down
不會出現在 Button-Up.

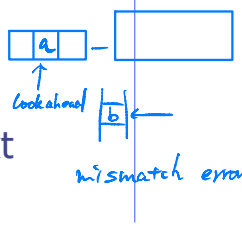
  - Where $A \rightarrow \alpha \mid \beta$ are two distinct productions in *G*, the following conditions hold:
    1. For no terminal a do both α and β derive strings beginning with a
    2. At most one of α and β can derive the empty string
    3. If β derives the empty string then α does not derive any string beginning with a terminal in FOLLOW(*A*)

$a \in FIRST(\alpha)$
$a \in FIRST(\beta)$
$\epsilon \in FOLLOW(A)$
$A \rightarrow \alpha \Rightarrow^{*} \epsilon$
$A \rightarrow \beta$

	a	b	c
A	A→α	A→β A→α	A→α

1. $\alpha \cap \beta = \phi$
2. $\alpha \Rightarrow \epsilon$, $\beta \not\Rightarrow \epsilon$
3. $FOLLOW(A) \cap FIRST(B) = \phi$

Error Handling in Predicting Parsing



- An error is detected when
  - The terminal on top of the stack does not match the next input symbol, or
  - Nonterminal *A* is on top of the stack, a is the next input symbol, and M[*A*, a] is empty
- Panic-mode error recovery
  - Based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears
  - Some heuristics of choosing the synchronizing tokens
    - Place all symbols in FOLLOW(*A*) into the synchronizing set of *A*
    - Add symbols in FIRST(*A*) to the synchronizing set of *A*
    - If a nonterminal can generate ϵ, then the production deriving ϵ can be used as a default
    - If a terminal on top of the stack cannot be matched, then pop the terminal

Recovery
$a_j = b + c$
1. panic mode
2. local correction
3. error production
4. global.

Table 標註 synchronize tokens

(5) → id = expr
1. FOLLOW(S)
2. FIRST(next line)

Error Handling in Predicting Parsing

- Example

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \epsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \epsilon$

  $F \rightarrow (E) \mid id$

- Parsing table
  - Fig. 4.22

- Actions
  - M[A, a] = Ø table 對應到空 input symbol is skipped
  - M[A, a] = sync nonterminal is popped
  - Token on stack isn't matched pop the token

sync

↳ 不要再丟了 後面要處理

Stack	Input	Output
$E	+ id * + id$	error, skip +
$E	id * + id$	id ∈ FIRST(E)
$E'T	id * + id$	
$E'T'F	id * + id$	
$E'T'id	id * + id$	
$E'T'	* + id$	
$E'T'F*	* + id$	
$E'T'F	+ id$	error, M[F,+] = sync
$E'T'	+ id$	F has been popped
$E'	+ id$	
$E'T+	+ id$	
$E'T	id$	
$E'T'F	id$	
$E'T'id	id$	
$E'T'	$	
$E'	$	
$	$	

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ id	synch	synch	$F \rightarrow (E)$	synch	synch

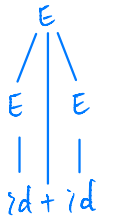
Bottom-Up Parsing

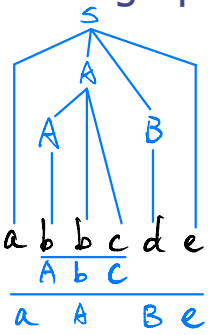
- Attempt to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards to the start symbol
  - Example. Consider the grammar

    S → aABe

    A → Abc | b

    B → d





handle b pos 2 Abc ↓ reduce

The sentence abbcde can be reduced to S:

reduce ↓

abbcde

aAbcde

aAde

aABe

S

Rightmost Derivation

S ⇒

aABe ⇒

aAde ⇒

aAbcde ⇒

abbcde

Bottom-Up Parsing

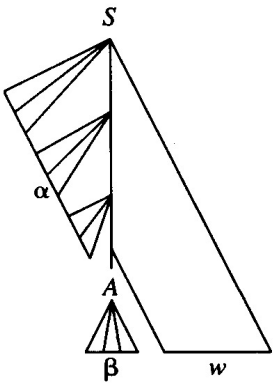
- A general style of bottom-up syntax analysis, known as *shift-reduce parsing,* will be introduced
  - The process of constructing a parse tree can be thought of as "reducing" a string *w* to the start symbol
  - It is equivalent to performing a rightmost derivation in reverse
  - At each reduction step, a particular substring matching the right side of a production is replaced by the symbol on the left
    - The matched substring can be called a *handle*
    - The reduction of a handle represents one step along the rightmost derivation in reverse
  - Formally, a handle of γ is
    - A production A → β, and
    - A position of γ where the substring β may be found
    - e.g. A → b at position 2 is a handle of abbcde
      A → Abc at position 2 is a handle of aAbcde

找 handle (位置)
替換

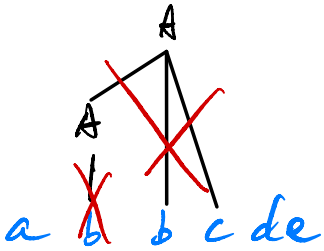
Handle Pruning

剪枝

- Suppose *A* → β is a handle
  - It represents a leftmost complete subtree consisting of a node and all its children
  - Reducing β to *A* in αβ*w* can be thought of "pruning the handle"
    - i.e. removing the children of *A*
- A rightmost derivation in reverse can be obtained by handle pruning

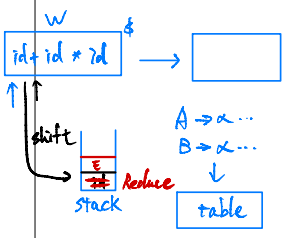


最後剩下 S 就對



Stack Implementation of Shift-Reduce Parsing

- Two problems for parsing by handle pruning
  - To locate the substring to be reduced
  - To determine what production to choose in case there is more than one production with that substring
- A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols
  - Actions
    - Shift
      - the next input symbol is shifted onto the top of the stack
    - Reduce
      - The handle is at the top of the stack
      - Remove the handle and place the left side nonterminal on the stack
    - Accept
      - Announce successful completion of parsing
    - error



Stack Implementation of Shift-Reduce Parsing

- Example. (1) $E \rightarrow E + E$
  (2) $E \rightarrow E * E$
  (3) $E \rightarrow (E)$
  (4) $E \rightarrow$ id

The precedence of * and + is not defined. Grammar is ambiguous.

Stack	Input	Action
$	id + id * id$	shift
$id	+ id * id$	reduce by (4)
$E	+ id * id$	shift
$E +	id * id$	shift
$E + id	* id$	reduce by (4)
$E + E	* id$	reduce by (1)
$E	* id$	shift
$E *	id$	shift
$E * id	$	reduce by (4)
$E * E	$	reduce by (2)
$E	$	accept

Stack	Input	Action
$	id + id * id$	shift
$id	+ id * id$	reduce by (4)
$E	+ id * id$	shift
$E +	id * id$	shift
$E + id	* id$	reduce by (4)
$E + E	* id$	shift
$E + E *	id$	shift
$E + E * id	$	reduce by (4)
$E + E * E	$	reduce by (2)
$E + E	$	reduce by (1)
$E	$	accept

Conflicts During Shift-Reduce Parsing

- For some grammars, a shift-reduce parser can't decide
  - whether to shift or to reduce (a *shift/reduce conflict*), or
  - which of several reductions to make (a *reduce/reduce conflict*)
  - Example 1

Stack	Input	Action	Stack	Input	Action
$E + E	* id$	shift	$E + E	* id$	reduce by $E \rightarrow E + E$
$E + E *	id$		$E	* id$	

  - Example 2

shift-reduce

stmt → if *expr* then *stmt*
| if *expr* then *stmt* else *stmt*
| other

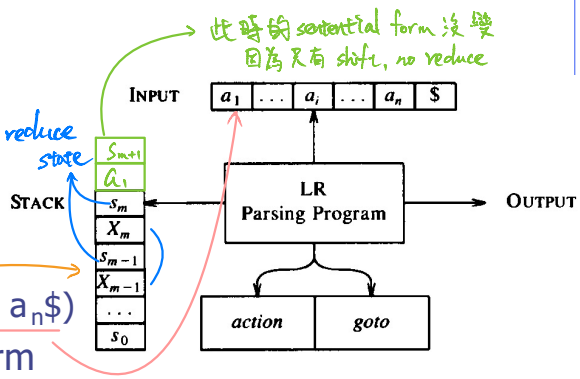
Stack	Input
... if *expr* then *stmt*	else ... $

LR Parsers

- LR(k) parsing LR(1) 就可以了 LALR(1) 在 LR(1) 中大部分 language 都可用
  - an efficient, bottom-up parsing technique
  - L: scanning the input from left to right
  - R: producing a rightmost derivation in reverse
  - k: the number of input symbols of lookahead
- Advantages
  - Can recognize virtually all programming-language constructs for which context-free grammars can be written
  - The most general nonbacktracking shift-reduce parsing → 因為有 parsing table
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers
  - Can detect a syntactic error as soon as possible
- Drawback LR parser 的 symbol table 難產生 → yacc
  - Too much work to construct an LR parser by hand

CPP is not LR(1) Because the class is too complicated can't be represented in nonterminal.

LR Parsing Algorithm



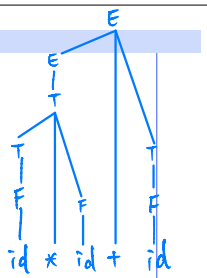
- Configuration
  - The stack contents
  - The unexpended input
    (s_0 X_1 s_1 X_2 s_2 … X_m s_m, a_i a_{i+1} … $a_n$$)
  - It represents the sentential form
    X_1X_2… $X_m a_i a_{i+1}$ … a_n
- Next move is determined by the input symbol a_i and stack top s_m
  - action[s_m, a_i] = shift s
    (s_0 X_1 s_1 X_2 s_2 … X_m s_m a_i s, a_{i+1} … $a_n$$)
  - action[s_m, a_i] = reduce $A \rightarrow \beta$
    (s_0 X_1 s_1 X_2 s_2 … X_{m-r} s_{m-r} A s, a_i … $a_n$$)
    where s = goto[s_{m-r}, A] and $|\beta|$ = r
  - action[s_m, a_i] = accept
  - action[s_m, a_i] = error

LR Parsing Algorithm

- Input. String w and an LR parsing table
- Output. A bottom-up parsing for w
- Method. Initially, stack = s_0 and input = $w$$

```
set ip to point to the first symbol of w$
repeat forever
  let s be the state on stack top and a the symbol pointed by ip
  if action[s, a] = shift s' then
    push a and s' onto the stack and advance the ip
  else if action[s, a] = reduce A → β then
    push 2*|β| symbols off the stack and now s' is the top state
    push A and then goto[s', A] on top of the stack
    output A → β
  else if action[s, a] = accept then
    return
  else error()
end
```

E → E+E | E*E | (E) | id ⟶ E → E+E | T
T → T*T | F
F → (E) | id



LR Parsing Algorithm

- Example.
  - (1) $E \rightarrow E + T$
  - (2) $E \rightarrow T$
  - (3) $T \rightarrow T * F$
  - (4) $T \rightarrow F$
  - (5) $F \rightarrow (E)$
  - (6) $F \rightarrow$ id
- Parsing table
  - Fig. 4.37

State	action					goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4		(accept)	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			

Stack	Input	Action
$0	id * id + id$	shift 5
$0 id 5	* id + id$	reduce by $F \rightarrow$ id
$0 F3	* id + id$	reduce by $T \rightarrow F$
$0 T2	* id + id$	shift
$0 T2 * 7	id + id$	shift
$0 T2 * 7 id 5	+ id$	reduce by $F \rightarrow$ id
$0 T2 * 7 F 10	+ id$	reduce by $T \rightarrow T * F$
$0 T2	+ id$	reduce by $E \rightarrow T$
$0 E1	+ id$	shift
$0 E1 + 6	id$	shift
$0 E1 + 6 id 5	$	reduce by $F \rightarrow$ id
$0 E1 + 6 F3	$	reduce by $T \rightarrow F$
$0 E1 + 6 T9	$	reduce by $E \rightarrow E + T$
$0 E1	$	accept

table is the most difficult part

Constructing LR Parsing Tables

- Three techniques for constructing an LR parsing table
  - Simple LR (SLR) (simple)
  - Canonical LR (LR) (complicated) 權威/標準 (少用因為太大)
  - Lookahead LR (LALR) (medium) yacc
- LR(0) item 不管 input 只看文法
  - A production of G with a dot at some position of the right side,
  - e.g. production $A \rightarrow XYZ$ yields
    - $A \rightarrow \cdot XYZ$ (fix pointer)
    - $A \rightarrow X \cdot YZ$
    - $A \rightarrow XY \cdot Z$
    - $A \rightarrow XYZ \cdot$

    (LR(0) item; $S' \rightarrow \cdot S$, $S' \rightarrow S \cdot$)

  Production $A \rightarrow \epsilon$ yields $A \rightarrow \cdot$

  $L(G) = \{w \mid S \Rightarrow w, w \in V_T\}$

  $G(V_N, V_T, P, S)$
- Augmented grammar $G'(V_N \cup S', V_T, P \cup \{S' \rightarrow S\}, S')$
  - G with a new start symbol S' and production $S' \rightarrow S$
  - Acceptance occurs when the parser is to reduce by $S' \rightarrow S \cdot$

也需要看 input ($) 判定結束

Closure Operation

- If I is a set of items, then *closure*(I) is the set of items constructed from I:
  - Initially, every item in I is added to *closure*(I)
  - If $A \rightarrow \alpha \cdot B\beta$ is in *closure*(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot\gamma$ to *closure*(I)
  - Repeat until no more new items can be added

2 operation
1. closure
2. goto

```
function closure(I)
  J = I
  repeat
    for each item A → α·Bβ ∈ J
    and each B → γ
    such that B → ·γ ∉ J do
        add B → ·γ to J
  until no more items can be added to J
  return J
end
```

- Example

$E' \rightarrow E$	$closure(\{E' \rightarrow \cdot E\}) = \{$
$E \rightarrow E + T$	$E' \rightarrow \cdot E$
$E \rightarrow T$	$E \rightarrow \cdot E + T$
$T \rightarrow T * F$	$E \rightarrow \cdot T$
$T \rightarrow F$	$T \rightarrow \cdot T * F$
$F \rightarrow (E)$	$T \rightarrow \cdot F$
$F \rightarrow$ id	$F \rightarrow \cdot (E)$
	$F \rightarrow \cdot$ id $\}$

Goto Operation

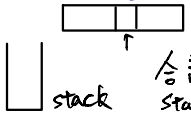
move dot from left to right

- If I is a set of items and X is a grammar symbol, then *goto*(I, X) is defined to be the closure of the set of all items $A \rightarrow \alpha X \cdot \beta$ such that $A \rightarrow \alpha \cdot X\beta$ is in I

可以是 terminal
也可以是 nT

Example

$E' \rightarrow E$	no "+", drop
$E \rightarrow E + T$	$goto(\{E' \rightarrow E \cdot, E \rightarrow E \cdot + T\}, +) =$
$E \rightarrow T$	$closure(\{E \rightarrow E + \cdot T\}) = \{$
$T \rightarrow T * F$	$E \rightarrow E + \cdot T$
$T \rightarrow F$	$T \rightarrow \cdot T * F$
$F \rightarrow (E)$	$T \rightarrow \cdot F$
$F \rightarrow$ id	$F \rightarrow \cdot (E)$
	$F \rightarrow \cdot$ id $\}$



合起來叫 sentential form
stack 裡的是 viable prefix

- Viable prefixes
  - The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser

Set-of-Items Construction

procedure *items*(*G′*)
  C = {*closure*({*S′* → · *S*})} I_0
  repeat
    for each set of items *I* in *C* and each grammar symbol *X*
    such that *goto*(*I*, *X*) is not empty and not in *C* do
      add *goto*(*I*, *X*) to *C*
  until no more set of items can be added to *C*
end

goto(I_0, E) = I_1
goto(I_0, T) = I_2
goto(I_0, F) = I_3
goto(I_0, '(') ⋮
goto(I_0, id)
看新的就叫 closure

Example *G* =
E′ → *E*
E → *E* + *T*
E → *T*
T → *T* * *F*
T → *F*
F → (*E*)
F → id

I_0: *E′* → · *E*
E → · *E* + *T*
E → · *T*
T → · *T* * *F*
T → · *F*
F → · (*E*)
F → · id

Set-of-Items Construction $C = \{I_0, I_1, \cdots I_{10}, I_{11}\}$

I_0: *E′* → · *E*
E → · *E* + *T*
E → · *T*
T → · *T* * *F*
T → · *F*
F → · (*E*)
F → · id

看到 NT 存

goto(I_0, *E*) =
I_1: *E′* → *E* ·
E → *E* · + *T*

ε-closure
goto

goto(I_0, *T*) =
I_2: *E* → *T* ·
T → *T* · * *F*

goto(I_0, *F*) =
I_3: *T* → *F* ·

goto(I_0, "(") =
I_4: *F* → (· *E*)
E → · *E* + *T*
E → · *T*
T → · *T* * *F*
T → · *F*
F → · (*E*)
F → · id

goto(I_0, id) =
I_5: *F* → id ·

goto(I_1, "+") =
I_6: *E* → *E* + · *T*
T → · *T* * *F*
T → · *F*
F → · (*E*)
F → · id

goto(I_2, "*") =
I_7: *T* → *T* * · *F*
F → · (*E*)
F → · id

goto(I_4, *E*) =
I_8: *F* → (*E* ·)
E → *E* · + *T*

goto(I_4, *T*) = I_2
goto(I_4, *F*) = I_3
goto(I_4, "(") = I_4
goto(I_4, id) = I_5

goto(I_6, *T*) =
I_9: *E* → *E* + *T* ·
T → *T* · * *F*

goto(I_6, *F*) = I_3
goto(I_6, "(") = I_4
goto(I_6, id) = I_5

goto(I_7, *F*) =
I_{10}: *T* → *T* * *F* ·

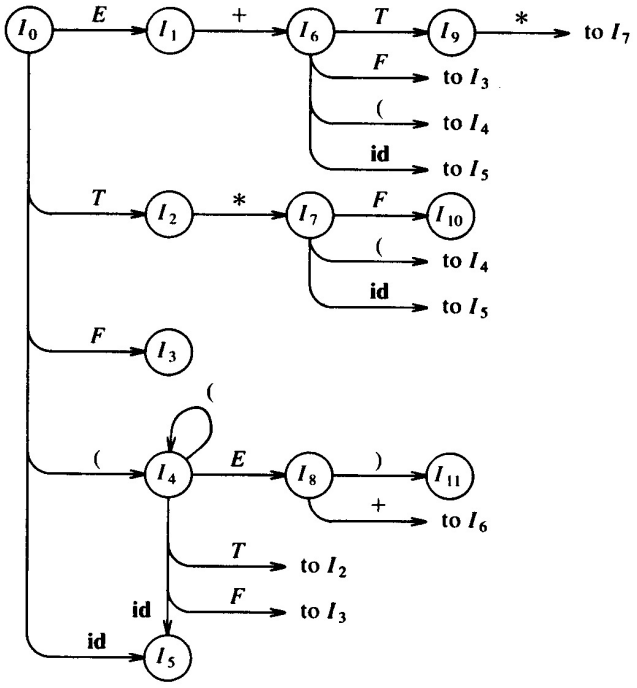
goto(I_7, "(") = I_4
goto(I_7, id) = I_5

goto(I_8, ")") =
I_{11}: *F* → (*E*) ·

goto(I_8, "+") = I_6
goto(I_9, "*") = I_7

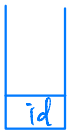
Transition Diagram of DFA for Viable Prefixes

- The *goto* function for the sets of items can be represented as the transition diagram of a DFA *D*
  - Each set of items I_i is a state of D



Rightmost Derivation

$E \Rightarrow$
$E + T \Rightarrow$
$E + F \Rightarrow$
$E +$ id $\Rightarrow$
$T +$ id $\Rightarrow$
$T * F +$ id $\Rightarrow$
$T *$ *id* + id $\Rightarrow$
$F *$ *id* + id $\Rightarrow$
id + id * id



Transition Diagram of DFA for Viable Prefixes

- *D* recognizes exactly the viable prefixes of *G* if
  - Each state is a final state, and
  - I_0 is the initial state
- For every grammar *G*, the *goto* function of the collection of sets of items defines a DFA that recognizes the viable prefix of *G*
  - If each item is treated as a state, an NFA *N* can be formed:
    - There is a transition from $A \rightarrow \alpha \cdot X \beta$ to $A \rightarrow \alpha X \cdot \beta$ label *X*, and
    - There is a transition from $A \rightarrow \alpha \cdot B \beta$ to $B \rightarrow \cdot \gamma$ labeled ϵ
  - Then *closure* (*I*) for the set of items *I* is exactly the ϵ-*closure* of a set of NFA states
  - Thus goto(*I*, *X*) gives the transition from *I* on symbol *X* in the DFA constructed from *N* by the subset construction
  - The procedure *items* (*G*′) is just the subset construction itself applied to the NFA *N*

Constructing an SLR Parsing Table

- Algorithm
  - Input. An augmented grammar G'
  - Output. The SLR parsing table for G'
  - Method.
    1. Construct $C = \{I_0, I_1, I_2, \ldots, I_n\}$, the sets of LR(0) items
    2. State i is constructed from I_i. Actions for state i
       - If $A \rightarrow \alpha \cdot a\beta \in I_i$ and goto(I_i, a) = I_j, then action[i, a] = shift j
       - If $A \rightarrow \alpha \cdot \in I_i$, then action[$i$, a] = reduce $A \rightarrow \alpha$ $\forall a \in$ FOLLOW(A)
       - If $S' \rightarrow S \cdot \in I_i$, then action[$i$, \$] = accept
    3. If goto(I_i, A) = I_j, then goto[i, A] = j
    4. The set of items containing $S' \rightarrow \cdot S$ is the initial state

Constructing an SLR Parsing Table

- Example
  (0) $E' \rightarrow E$
  (1) $E \rightarrow E + T$
  (2) $E \rightarrow T$
  (3) $T \rightarrow T * F$
  (4) $T \rightarrow F$
  (5) $F \rightarrow (E)$
  (6) $F \rightarrow$ id

- FOLLOW(E) ={+,), \$}
- FOLLOW(T) = {+, *,), \$}
- FOLLOW(F) = {+, *,), \$}

$C = \{I_0, I_1, I_2, \ldots I_{11}\}$

	action V_T						goto V_N		
	id	+	*	(	)	\$	E	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			

reduce or shift　　move to next state

1. goto(I_i, a) = I_j
   action(i, a) = shift j
2. $A \rightarrow \alpha \cdot \in I_i$
   action[i, a] = reduce $A \rightarrow \alpha$
3. goto(I_i, A) = I_i
   goto(i, A) = j
4. $S' \rightarrow S \cdot$
   action[i, \$] = accept

Constructing an SLR Parsing Table

- Some grammars are not SLR, e.g.

(0) $S' \rightarrow S$
(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow *R$
(4) $L \rightarrow$ id
(5) $R \rightarrow L$

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot *R$
$L \rightarrow \cdot$ id
$R \rightarrow \cdot L$

$goto(I_0, S) =$
I_1: $S' \rightarrow S \cdot$

$goto(I_0, L) =$
I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

$goto(I_0, R) =$
I_3: $S \rightarrow R \cdot$

$goto(I_0, \text{"*"}) =$
I_4: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot$ id

$goto(I_0,$ id$) =$
I_5: $L \rightarrow$ id $\cdot$

$goto(I_2, \text{"="}) =$
I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot$ id

$goto(I_4, R) =$
I_7: $L \rightarrow *R \cdot$

$goto(I_4, L) =$
I_8: $R \rightarrow L \cdot$

$goto(I_4, \text{"*"}) = I_4$

$goto(I_4,$ id$) = I_5$

$goto(I_6, R) =$
I_9: $S \rightarrow L = R \cdot$

$goto(I_6, L) = I_8$

$goto(I_6, \text{"*"}) = I_4$

$goto(I_6,$ id$) = I_5$

Constructing an SLR Parsing Table

- Some grammars are not SLR, e.g.

(0) $S' \rightarrow S$
(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow * R$
(4) $L \rightarrow$ id
(5) $R \rightarrow L$

- FOLLOW(S) ={$}
- FOLLOW(L) = {=, $}
- FOLLOW(R) = {=, $}

集合是{=, $}

I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

⇒ 此語言不屬於SLR(1)

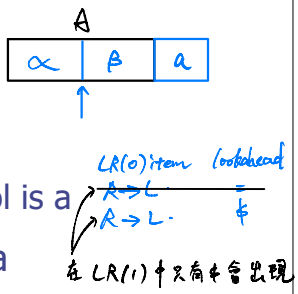
	action				goto		
	id	=	*	$	S	L	R
0							
1							
2		s6 r5	conflict				
3							
4							
5							
6							
7							
8							
9							

id = id
reduce or shift ?

LR(0) item | lookahead
$R \rightarrow L \cdot$ | =
$R \rightarrow L \cdot$ | $
→ reduce
→ reduce

Constructing Canonical LR Parsing Tables

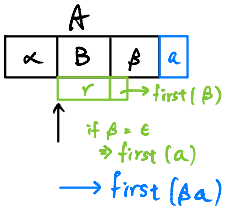
- Why does SLR sometimes fail
  - $R \rightarrow L \cdot \in I_2 \Rightarrow$ action[2, '='] = reduce $R \rightarrow L$
    $\because = \in$ FOLLOW(R)
  - However, there is no sentential form begins $R =$ ...
  - Extra information will be incorporated by redefining items
- LR(1) items
  - The general form of an items becomes $[A \rightarrow \alpha \cdot \beta, a]$
    - The lookahead a has no effect when $\beta \neq \epsilon$
    - $[A \rightarrow \alpha \cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ if next symbol is a
  - Formally, LR(1) is valid for a viable prefix γ if there is a derivation $S \Rightarrow_{rm}^{*} \delta Aw \Rightarrow \delta\alpha\beta w$, where
    - $\gamma = \delta\alpha$, and
    - Either a is the first symbol of w, or w is ϵ and a is $



Sets of LR(1) Items

function *closure* (*I*)
  J = *I*
  repeat
    for each item $[A \rightarrow \alpha \cdot B\beta, a] \in J$
    each $B \rightarrow \gamma$ in G′ and
    each terminal b ∈ FIRST(βa)
    such that $[B \rightarrow \cdot \gamma, b] \notin J$ do
      add $[B \rightarrow \cdot \gamma, b]$ to *J*
  until no more items can be added to *J*
  return *J*
end

function *goto* (*I*, *X*)
  J = set of items $[A \rightarrow \alpha X \cdot \beta, a]$
    such that $[A \rightarrow \alpha \cdot X\beta, a] \in I$
  return *closure* (*J*)
end



procedure *items* (*G*′)
  C = {*closure* ({[*S*′ → · *S*, $]})}
  repeat
    for each set of item $I \in C$ and each grammar symbol *X*
    such that *goto* (*I*, *X*) ∉ *C* do
      add *goto* (*I*, *X*) to *C*
  until no more sets of items can be added to *C*
end

Sets of LR(1) Items

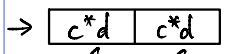
Example G'=

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$



I_0: $S' \rightarrow \cdot S$, \$

$S \rightarrow \cdot\, CC$, \$ FIRST(\$)

$C \rightarrow \cdot\, cC$, c/d FIRST(C\$)

$C \rightarrow \cdot\, d$, c/d

$A \rightarrow \alpha \cdot B\beta, a$

$[A \rightarrow \alpha \cdot B\,\beta, a]$

goto (I_0, S) =
I_1: $S' \rightarrow S \cdot$, \$

goto (I_0, d) =
I_4: $C \rightarrow d \cdot$, c/d

goto (I_2, d) =
I_7: $C \rightarrow d \cdot$, \$

goto (I_0, C) =
I_2: $S \rightarrow C \cdot C$, \$
$C \rightarrow \cdot\, cC$, \$
$C \rightarrow \cdot\, d$, \$

goto (I_2, C) =
I_5: $S \rightarrow CC \cdot$, \$

goto (I_3, C) =
I_8: $C \rightarrow cC \cdot$, c/d

goto (I_3, c) = I_3

goto (I_3, d) = I_4

goto (I_0, c) =
I_3: $C \rightarrow c \cdot C$, c/d
$C \rightarrow \cdot\, cC$, c/d
$C \rightarrow \cdot\, d$, c/d

goto (I_2, c) =
I_6: $C \rightarrow c \cdot C$, \$
$C \rightarrow \cdot\, cC$, \$
$C \rightarrow \cdot\, d$, \$

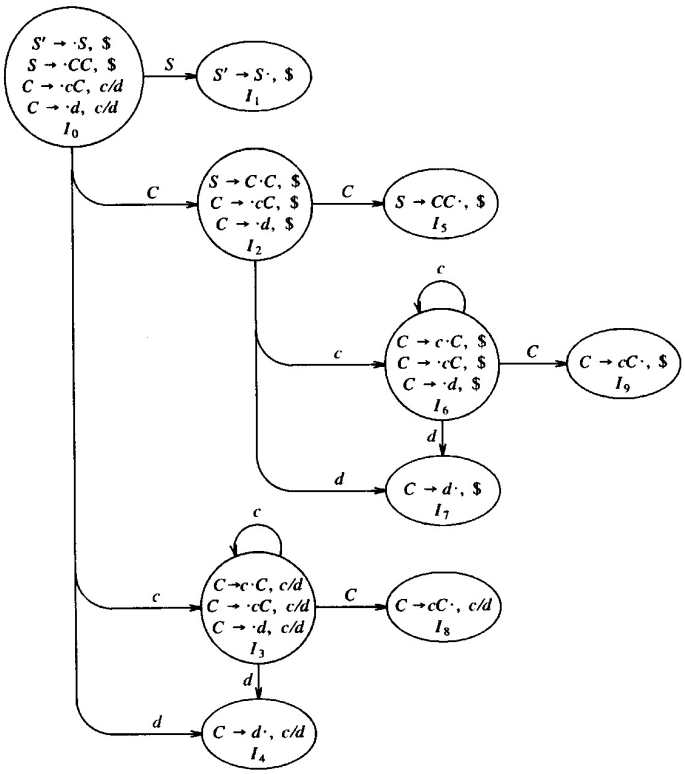
goto (I_6, C) =
I_9: $C \rightarrow cC \cdot$, \$

goto (I_6, c) = I_6

goto (I_6, d) = I_7

Sets of LR(1) Items

The *goto* graph



Construction of the LR Parsing Table

- Algorithm
  - Input. An augmented grammar G'
  - Output. The canonical LR parsing table for G'
  - Method.
    1. Construct $C = \{I_0, I_1, I_2, \ldots, I_n\}$, the sets of LR(1) items
    2. State i is constructed from I_i. Actions for state i
       - If $[A \rightarrow \alpha \cdot a\beta, b] \in I_i$ and $goto(I_i, a) = I_j$, then action[i, a] = shift j
       - If $[A \rightarrow \alpha \cdot, a] \in I_i$ and , $A \neq S'$ then action[i, a] = reduce $A \rightarrow \alpha$
       - If $[S' \rightarrow S \cdot, \$] \in I_i$, then action[$i$, \$] = accept
    3. If $goto(I_i, A) = I_j$, then goto[i, A] = j
    4. The set of items containing $[S' \rightarrow \cdot S, \$]$ is the initial state

Constructing an LR Parsing Table

Example G'=

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

$goto(I_6, d) = I_7$

SLR 和 LR state 差 5~10 倍

	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Constructing LALR Parsing Tables

- Lookahead LR (LALR) technique
  - Often used in practice because the tables are much smaller than the LR tables
  - Yet most common syntactic constructs of programming languages can be expressed by an LALR grammar
  - Consider the following sets of LR(1) items

I_3: $C \to c \cdot C$, c/d
$C \to \cdot cC$, c/d
$C \to \cdot d$, c/d

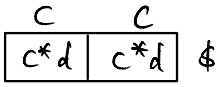
I_6: $C \to c \cdot C$, $
$C \to \cdot cC$, $
$C \to \cdot d$, $

I_4: $C \to d \cdot$, c/d (core)

I_7: $C \to d \cdot$, $

I_8: $C \to cC \cdot$, c/d

I_9: $C \to cC \cdot$, $



SLR 單純根據 follow

LALR：找相同的 core，合併

$I_3, I_6 \Rightarrow I_{36}$

$I_4, I_7 \Rightarrow I_{47}$: {(c→d, c/d/$)}

  - The only difference is the lookahead symbols
  - Note the G' generates the language c*dc*d
  - An LALR parsing table can be obtained by merging the sets of items of a LR table with the same *core*

Construction of the LALR Parsing Table

- Algorithm
  - Input. An augmented grammar G'
  - Output. The LALR parsing table for G'
  - Method.
    1. Construct $C = \{I_0, I_1, I_2, \ldots, I_n\}$, the sets of LR(1) items
    2. For each core in C, find all sets with that core and replace these sets by their union. Let result be $C' = \{J_0, J_1, J_2, \ldots, J_m\}$
    3. State i is constructed from J_i. Actions for state i
       - If $[A \to \alpha \cdot a\beta, b] \in J$ and goto(J_i, a) = J_j, then action[i, a] = shift j
       - If $[A \to \alpha \cdot, a] \in J_i$ and , $A \neq S'$ then action[i, a] = reduce $A \to \alpha$
       - If $[S' \to S \cdot, \$] \in J_i$, then action[$i$, $] = accept
    4. If $J_i = I_1 \cup I_2 \cup \ldots \cup I_k$, then
       $goto(I_1, X) = goto(I_2, X) = \ldots = goto(I_k, X) = J_j$
       $\Rightarrow$ goto[i, X] = j

Construction of the LALR Parsing Table

I_3: $C \to c \cdot C$, c/d
$C \to \cdot cC$, c/d
$C \to \cdot$ d, c/d

I_6: $C \to c \cdot C$, \$
$C \to \cdot cC$, \$
$C \to \cdot$ d, \$

I_4: $C \to$ d $\cdot$, c/d

I_7: $C \to$ d $\cdot$, \$

I_8: $C \to cC \cdot$, c/d

I_9: $C \to cC \cdot$, \$

$C' = \{I_0, I_1, I_2, I_{36}, I_{47}, I_5, I_{89}\}$

	action			goto	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	action			goto	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
6	s6	s7			9
7			r3		
89	r2	r2	r2		
9			r2		

Error Detection by LALR

- LALR parser may proceed to do some reductions after the LR parser has declared an error
  - LALR parser will never shift another symbol after the LR parser declares an error
  - Example w = ccd\$

	action			goto	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
6	s6	s7			9
7			r3		
89	r2	r2	r2		
9			r2		

LALR

Stack	Input	Action
\$0	ccd\$	shift 36
\$0 c 36	cd\$	shift 36
\$0 c 36 c 36	d\$	shift ~~36~~ 47
\$0 c 36 c 36 d 47	\$	reduce by $C \to$ d
\$0 c 36 c 36 C 89	\$	reduce by $C \to cC$
\$0 c 36 C 89	\$	reduce by $C \to cC$
\$0 C 2	\$	error

LR

Stack	Input	Action
\$0	ccd\$	shift 3
\$0 c 3	cd\$	shift 3
\$0 c 3 c 3	d\$	shift ~~3~~ 4
\$0 c 3 c 3 d 4	\$	error

always find error at the same point.
but LALR need more reduction steps.

Conflicts Caused by Merging States

- Merging states with common cores might cause conflicts
  - No shift-reduce conflicts will be introduced
    - Because shift actions depend only on the core, not the lookahead
  - Reduce-reduce conflicts might be produced by merging

Example G' =
$S' \rightarrow S$
$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
$A \rightarrow c$
$B \rightarrow c$

goto (I_0, a) =
I_2: $S \rightarrow a \cdot Ad \mid a \cdot Be$, \$
$A \rightarrow \cdot c$, d
$B \rightarrow \cdot c$, e

goto (I_2, c) =
I_4: $A \rightarrow c \cdot$, d
$B \rightarrow c \cdot$, e

I_0: $S' \rightarrow \cdot S$, \$
$S \rightarrow \cdot aAd \mid \cdot bBd \mid \cdot aBe \mid \cdot bAe$, \$

goto (I_0, S) = I_1: $S' \rightarrow S \cdot$, \$

goto (I_0, b) =
I_3: $S \rightarrow b \cdot Bd \mid b \cdot Ae$, \$
$A \rightarrow \cdot c$, e
$B \rightarrow \cdot c$, d

goto (I_3, c) =
I_5: $A \rightarrow c \cdot$, e
$B \rightarrow c \cdot$, d

	d	e
45	rA→c rB→c	rA→c rB→c

merge I_4 and I_5 =
I_{45}: $A \rightarrow c \cdot$, d/e
$B \rightarrow c \cdot$, d/e

Using Ambiguous Grammars

- Every ambiguous grammar fails to be LR
  - Certain types of ambiguous grammars are useful in the specification and implementation of languages
    - An ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar
      e.g. $E \rightarrow E + E \mid E * E \mid (E) \mid$ id 加1規則，2優先順序
    - Ambiguous grammars can be used to isolate commonly occurring syntactic constructs for special case optimization
  - Disambiguating rules must be specified to allow only one parse tree for each sentence
    - e.g. using precedence and associativity to resolve conflicts
    - In this way, the overall language specification still remains unambiguous

Top-Down LL(1)
Button-Up SLR(1) LALR(1) LR(1)

Using Precedence and Associativity

E+E*E

Example G'
(0) $E' \rightarrow E$
(1) $E \rightarrow E + E$
(2) $E \rightarrow E * E$
(3) $E \rightarrow (E)$
(4) $E \rightarrow$ id

I_0: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot$ id

goto (I_0, E) =
I_1: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

goto (I_0, '(') =
I_2: $E \rightarrow (\cdot E)$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot$ id

goto (I_0, id) =
I_3: $E \rightarrow$ id $\cdot$

goto (I_1, '+') =
I_4: $E \rightarrow E + \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot$ id

goto (I_1, '*') =
I_5: $E \rightarrow E * \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot$ id

goto (I_2, E) =
I_6: $E \rightarrow (E \cdot)$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

goto (I_4, E) =
I_7: $E \rightarrow E + E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

% left '*'
% left '+'
left-association

goto (I_5, E) =
I_8: $E \rightarrow E * E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

goto (I_6, ')') =
I_9: $E \rightarrow (E) \cdot$

	action			goto
	+	*	...	
...				
7	r1	s5		
8	r2	r2		
9				

The "Dangling-else" Ambiguity

if else

Example G'
(0) $S' \rightarrow S$
(1) $S \rightarrow$ i S e S
(2) $S \rightarrow$ i S
(3) $S \rightarrow$ a

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot$ i S e S
$S \rightarrow \cdot$ i S
$S \rightarrow \cdot$ a

goto (I_0, S) =
I_1: $S' \rightarrow S \cdot$

goto (I_0, i) =
I_2: $S \rightarrow$ i $\cdot S$ e S
$S \rightarrow$ i $\cdot S$
$S \rightarrow \cdot$ i S e S
$S \rightarrow \cdot$ i S
$S \rightarrow \cdot$ a

goto (I_0, a) =
I_3: $S \rightarrow$ a $\cdot$

goto (I_2, S) =
I_4: $S \rightarrow$ i $S \cdot$ e S 應該要shift
$S \rightarrow$ i $S \cdot$

goto (I_4, e) =
I_5: $S \rightarrow$ i S e $\cdot S$
$S \rightarrow \cdot$ i S e S
$S \rightarrow \cdot$ i S
$S \rightarrow \cdot$ a

goto (I_5, S) =
I_6: $S \rightarrow$ i S e $S \cdot$

	action			goto
	i	e	...	
...				
4		s5		
8				
9				

yacc default:
shift-reduce conflict
都是shift，符合我們的Project，因此不需更改

Ambiguities from Special-Case Productions

Example G'

(0) $E' \rightarrow E$
(1) $E \rightarrow E$ sub E sup E
(2) $E \rightarrow E$ sub E
(3) $E \rightarrow E$ sup E
(4) $E \rightarrow \{E\}$
(5) $E \rightarrow$ c

% right sub sup

I_0: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E$ sub E sup E
$E \rightarrow \cdot E$ sub E
$E \rightarrow \cdot E$ sup E
$E \rightarrow \cdot \{E\}$
$E \rightarrow \cdot$ c

I_7: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot$ sub E sup E
$E \rightarrow E$ sub $E \cdot$ sup E
$E \rightarrow E \cdot$ sub E
$E \rightarrow E$ sub $E \cdot$
$E \rightarrow E \cdot$ sup E

I_8: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot$ sub E sup E
$E \rightarrow E \cdot$ sub E
$E \rightarrow E \cdot$ sup E
$E \rightarrow E$ sup $E \cdot$

I_{11}: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot$ sub E sup E
$E \rightarrow E$ sub E sup $E \cdot$
$E \rightarrow E \cdot$ sub E
$E \rightarrow E \cdot$ sup E
$E \rightarrow E$ sup $E \cdot$

right associated ⇒ shift

	action				goto
	sub	sup	}	$	
...					
7	s4 r2	s10 r2	r2	r2	
8	s4 r3	s5 r3	r3	r3	
9					
11	s4 r1 r3	s5 r1 r3	r1 r3	r1 r3	

reduce - reduce conflict
保留長的，因為長的是 special case

Operator-Precedence Parsing

- Works on a class of grammars: *operator grammars*
  - No production right side is ϵ or has two adjacent nonterminals
  - e.g. $E \rightarrow EAE \mid (E) \mid -E \mid$ id
    $A \rightarrow + \mid - \mid * \mid /$
    is not an operator grammar
  - $E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid -E \mid$ id
    is an operator grammar
- Advantages
  - Easy to implement
- Disadvantages
  - Hard to handle tokens like the minus sign (with 2 precedences)
  - Can't be sure the parser accepts exactly the desired language
  - Only a small class of grammars can be parsed

E + E
E op E ⇒ 這不是 operator grammar
* •> +
precedence 的大於

Operator-Precedence Parsing

- Precedence relations

Relation	Meaning
a <· b	a "yields precedence to" b
a ≐ b	a "has the same precedence as" b
a ·> b	a "takes precedence over" b

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

	id	+	*	$
id	不會碰到	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

right

left

Operator-Precedence Parsing

- The handle can be found by the steps
  - Scan the string from left until the first ·> is encountered
  - Then scan back backwards over any ≐ until a <· is encountered
  - The handle contains everything within the <· and ·> above, including any intervening or surrounding nonterminals
  - e.g. w = id + id * id

$ <· id ·> + <· id ·> * <· id ·> $ 1. skip ≐ ·< , stop by ·>

$ <· E + <· id ·> * <· id ·> $ 2. look backward to find ·<

$ <· E + <· E * <· id ·> $ ⇒ handle

$ <· E + <· E * E ·> $

$ <· E + E ·> $

$E $