LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

one input symbol used as a look-ahead symbol do determine parser action

LL(1) left most derivation

input scanned from left to right

- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.

A Grammar which is not LL(1)

$S \rightarrow i\ C\ t\ S\ E \quad | \quad a$

$E \rightarrow e\ S \quad | \quad \varepsilon$

$C \rightarrow b$

FOLLOW(S) = { \$,e }

FOLLOW(E) = { \$,e }

FOLLOW(C) = { t }

FIRST(S) = {i, a}

FIRST(E) = {e, ε}

FIRST(C) = {b}

	a	**b**	**e**	**i**	**t**	**\$**
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow e\ S$ $E \rightarrow \varepsilon$			$E \rightarrow \varepsilon$
C		$C \rightarrow b$				

two production rules for M[E,e]

Problem ➔ ambiguity

A Grammar which is not LL(1) (cont.)

- What do we have to do it if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$
    - ➔ any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.
    - ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$

1. Both α and β cannot derive strings starting with same terminals.

2. At most one of α and β can derive to ε.

3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
  - if the terminal symbol on the top of stack does not match with the current input symbol.
  - if the top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A,a] is empty.
- What should the parser do in an error case?
  - The parser should be able to give an error message (as much as possible meaningful error message).
  - It should be recover from that error case, and it should be able to continue the parsing with the rest of the input.

Error Recovery Techniques

- Panic-Mode Error Recovery
  - Skipping the input symbols until a synchronizing token is found.
- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- Error-Productions
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
  - When an error production is used by the parser, we can generate appropriate error diagnostics.
  - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- Global-Correction
  - Ideally, we would like a compiler to make as few change as possible in processing incorrect inputs.
  - We have to globally analyze the input to find the error.
  - This is an expensive method, and it is not in practice.

Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.
- What is the synchronizing token?
  - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.
- So, a simple panic-mode error recovery for the LL(1) parsing:
  - All the empty entries are marked as ***synch*** to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A which on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
  - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

Panic-Mode Error Recovery - Example

$S \rightarrow AbS \mid e \mid \varepsilon$
$A \rightarrow a \mid cAd$

FOLLOW(S)={\$}
FOLLOW(A)={b,d}

	a	**b**	**c**	**d**	**e**	**\$**
S	$S \rightarrow AbS$		$S \rightarrow AbS$		$S \rightarrow e$	$S \rightarrow \varepsilon$
A	$A \rightarrow a$	***sync***	$A \rightarrow cAd$	***sync***		

stack	**input**	**output**
\$S	aab\$	$S \rightarrow AbS$
\$SbA	aab\$	$A \rightarrow a$
\$Sba	aab\$	
\$Sb	ab\$	Error: missing b, inserted
\$S	ab\$	$S \rightarrow AbS$
\$SbA	ab\$	$A \rightarrow a$
\$Sba	ab\$	
\$Sb	b\$	
\$S	\$	$S \rightarrow \varepsilon$
\$	\$	accept

stack	**input**	**output**
\$S	ceadb\$	$S \rightarrow AbS$
\$SbA	ceadb\$	$A \rightarrow cAd$
\$SbdAc	ceadb\$	
\$SbdA	eadb\$	Error:unexpected e (illegal A)
(Remove all input tokens until first b or d, pop A)		
\$Sbd	db\$	
\$Sb	b\$	
\$S	\$	$S \rightarrow \varepsilon$
\$	\$	accept

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - change, insert, or delete input symbols.
  - issue appropriate error messages
  - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

How to select synchronizing set?

- Place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it likely that parsing can continue.
- We might add keywords that begins statements to the synchronizing sets for the nonterminals generating expressions.

How to select synchronizing set? (II)

- If a nonterminal can generate the empty string, then the production deriving ε can be used as a default. This may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
- If a terminal on top of stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted.

Example: error recovery

"synch" indicating synchronizing tokens obtained from FOLLOW set of the nonterminal in question.

If the parser looks up entry M[A,a] and finds that it is blank, the input symbol a is skipped.

If the entry is synch, the the nonterminal on top of the stack is popped.

If a token on top of the stack does not match the input symbol, then we pop the token from the stack.

$$\mathrm{FIRST}(E) = \mathrm{FIRST}(T) = \mathrm{FIRST}(F) = \{(, \mathbf{id}\}.$$

$$\mathrm{FIRST}(E') = \{+, \epsilon\}$$

$$\mathrm{FIRST}(T') = \{*, \epsilon\}$$

$$\mathrm{FOLLOW}(E) = \mathrm{FOLLOW}(E') = \{),\ \$\}$$

$$\mathrm{FOLLOW}(T) = \mathrm{FOLLOW}(T') = \{+,), \$\}$$

$$\mathrm{FOLLOW}(F) = \{+, *,), \$\}$$

NONTER-MINAL	INPUT SYMBOL id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$	synch	synch	$F \to (E)$	synch	synch

Fig. 4.18. Synchronizing tokens added to parsing table of Fig. 4.15.

Example: error recovery (II)

STACK	INPUT	REMARK
$\$E$	$)\,\mathbf{id} * + \mathbf{id}\,\$$	error, skip)
$\$E$	$\mathbf{id} * + \mathbf{id}\,\$$	**id** is in FIRST(E)
$\$E'T$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\$E'T'F$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\$E'T'\mathbf{id}$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\$E'T'$	$* + \mathbf{id}\,\$$	
$\$E'T'F*$	$* + \mathbf{id}\,\$$	
$\$E'T'F$	$+ \mathbf{id}\,\$$	error, $M[F, +]$ = synch
$\$E'T'$	$+ \mathbf{id}\,\$$	F has been popped
$\$E'$	$+ \mathbf{id}\,\$$	
$\$E'T+$	$+ \mathbf{id}\,\$$	
$\$E'T$	$\mathbf{id}\,\$$	
$\$E'T'F$	$\mathbf{id}\,\$$	
$\$E'T'\mathbf{id}$	$\mathbf{id}\,\$$	
$\$E'T'$	$\$$	
$\$E'$	$\$$	
$\$$	$\$$	

Fig. 4.19. Parsing and error recovery moves made by predictive parser.