COMPILER DESIGN

LL1 Grammar

Session Objectives

- Learn the concepts of LL1 Grammar
- Error recovery methods of predictive parsing

Session Outcomes

- At the end of this session, participants will be able to
  - Identify the LL1 grammar
  - Perform error recovery in predictive parsing

Agenda

- LL1 grammar
- Error recovery in predictive parsing

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-head 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.

LL(1) Grammars are Unambiguous

Ambiguous grammar
$S \rightarrow$ **i** E **t** S S' | **a**
$S' \rightarrow$ **e** S | ε
$E \rightarrow$ **b**

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$S \rightarrow$ **i** E **t** S S'	**i**	**e $**
$S \rightarrow$ **a**	**a**	**e $**
$S' \rightarrow$ **e** S	**e**	**e $**
$S' \rightarrow \varepsilon$	ε	**e $**
$E \rightarrow$ **b**	**b**	**t**

Error: duplicate table entry

	a	b	e	i	t	$
S	$S \rightarrow$ **a**			$S \rightarrow$ **i** E **t** S S'		
S'			$S' \rightarrow \varepsilon$ $S' \rightarrow$ **e** S			$S' \rightarrow \varepsilon$
E		$E \rightarrow$ **b**				

- 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 which is not LL(1)

- 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 $A \rightarrow \beta$

  - Both α and β cannot derive strings starting with same terminals.
  - At most one of α and β can derive to ε.
  - If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

Error Recovery - 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 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 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.

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	$S \rightarrow AbS$	*sync*	$S \rightarrow AbS$	*sync*	$S \rightarrow e$	$S \rightarrow \varepsilon$
A	$A \rightarrow a$	*sync*	$A \rightarrow cAd$	*sync*	*sync*	*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
$S	ceadb$	$S \rightarrow AbS$
$SbA	ceadb$	$A \rightarrow cAd$
$SbdAc	ceadb$	
$SbdA	eadb$	unexpected e (illegal A)

(Remove all input tokens until first b or d, pop A)

stack	input	output
$Sbd	db$	
$Sb	b$	
$S	$	$S \rightarrow \varepsilon$
$	$	accept

Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

Pro:	Can be automated
Cons:	Error messages are needed

FOLLOW(E) = { **)** **$** }
FOLLOW(E') = { **)** **$** }
FOLLOW(T) = { **+** **)** **$** }
FOLLOW(T') = { **+** **)** **$** }
FOLLOW(F) = { **+** ***** **)** **$** }

	id	**+**	*****	**(**	**)**	**$**
E	$E \to T E'$			$E \to T E'$	***synch***	***synch***
E'		$E' \to + T E'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to F T'$	***synch***		$T \to F T'$	***synch***	***synch***
T'		$T' \to \varepsilon$	$T' \to * F T'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ **id**	***synch***	***synch***	$F \to$ **(** E **)**	***synch***	***synch***

synch: the driver pops current nonterminal A and skips input till synch token or skips input until one of FIRST(A) is found

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.

Phrase-Level Recovery

Change input stream by inserting missing tokens
For example: **id id** is changed into **id * id**

Pro:	Can be automated
Cons:	Recovery not always intuitive

Can then continue here

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow T\,E'$			$E \rightarrow T\,E'$	***synch***	***synch***
E'		$E' \rightarrow \mathbf{+}\,T\,E'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\,T'$	***synch***		$T \rightarrow F\,T'$	***synch***	***synch***
T'	***insert ****	$T' \rightarrow \varepsilon$	$T' \rightarrow \mathbf{*}\,F\,T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$	***synch***	***synch***	$F \rightarrow \mathbf{(}\,E\,\mathbf{)}$	***synch***	***synch***

insert *: driver inserts missing ***** and retries the production

Error Productions

$E \to T\,E'$
$E' \to \mathbf{+}\,T\,E' \mid \varepsilon$
$T \to F\,T'$
$T' \to \mathbf{*}\,F\,T' \mid \varepsilon$
$F \to \mathbf{(}\,E\,\mathbf{)} \mid \mathbf{id}$

Add "error production":

$T' \to F\,T'$

to ignore missing *****, e.g.: **id id**

Pro:	Powerful recovery method
Cons:	Cannot be automated

	id	**+**	*****	**(**	**)**	**$**
E	$E \to T\,E'$			$E \to T\,E'$	***synch***	***synch***
E'		$E' \to \mathbf{+}\,T\,E_R$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to F\,T'$	***synch***		$T \to F\,T'$	***synch***	***synch***
T'	$T' \to F\,T'$	$T' \to \varepsilon$	$T' \to \mathbf{*}\,F\,T'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \mathbf{id}$	***synch***	***synch***	$F \to \mathbf{(}\,E\,\mathbf{)}$	***synch***	***synch***

Summary

- LL(1) Grammar
- Error recovery in predictive parsing

Check your understanding?

How the error during parsing the strings **aab** and **ceadb** will be handled in top down parser for the following grammar?

$S \rightarrow AbS \mid e \mid \varepsilon$

$A \rightarrow a \mid cAd$