



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 to 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.

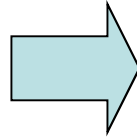
LL(1) Grammars are Unambiguous

Ambiguous grammar

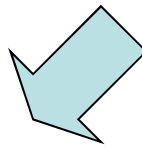
$S \rightarrow i E t S S' \mid a$

$S' \rightarrow e S \mid \epsilon$

$E \rightarrow b$



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

- What do we have to do 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 $\text{FIRST}(\beta)$ also appears $\text{FIRST}(A\alpha)$ because $A\alpha \Rightarrow \beta\alpha$.
 - ➔ If β is ϵ , any terminal that appears in $\text{FIRST}(\alpha)$ also appears in $\text{FIRST}(A\alpha)$ and $\text{FOLLOW}(A)$.

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 $\text{FIRST}(\alpha\beta_1)$ also appears in $\text{FIRST}(\alpha\beta_2)$.
- 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 analyze¹² 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

$S \rightarrow AbS \mid e \mid \epsilon$

$A \rightarrow a \mid cAd$

$\text{FOLLOW}(S) = \{\$ \}$

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

	a	b	c	d	e	\$
S	$S \rightarrow AbS$	<i>sync</i>	$S \rightarrow AbS$	<i>sync</i>	$S \rightarrow e$	$S \rightarrow \epsilon$
A	$A \rightarrow a$	<i>sync</i>	$A \rightarrow cAd$	<i>sync</i>	<i>sync</i>	<i>sync</i>

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 \$ $S \rightarrow \epsilon$

\$ \$ accept

stack input output

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)

\$Sbd db\$

\$Sb b\$

\$S \$ $S \rightarrow \epsilon$

\$ \$ accept

Panic Mode Recovery

Add synchronizing actions to
undefined entries based on FOLLOW

Pro:	Can be automated
Cons:	Error messages are needed

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

	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	<i>synch</i>	<i>synch</i>
E'		$E' \rightarrow + TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	<i>synch</i>		$T \rightarrow FT'$	<i>synch</i>	<i>synch</i>
T'		$T' \rightarrow \epsilon$	$T' \rightarrow * FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$	<i>synch</i>	<i>synch</i>	$F \rightarrow (E)$	<i>synch</i>	<i>synch</i>

synch: the driver pops current nonterminal A and skips input till
synch token or skips input until one of $\text{FIRST}(A)$ 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	$E \rightarrow T E'$			$E \rightarrow T E'$	<i>synch</i>	<i>synch</i>
E'		$E' \rightarrow + T E'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow F T'$	<i>synch</i>		$T \rightarrow F T'$	<i>synch</i>	<i>synch</i>
T'	<i>insert *</i>	$T' \rightarrow \epsilon$	$T' \rightarrow * F T'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$	<i>synch</i>	<i>synch</i>	$F \rightarrow (E)$	<i>synch</i>	<i>synch</i>

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

Error Productions

$E \rightarrow T E'$
 $E' \rightarrow + T E' \mid \epsilon$
 $T \rightarrow F T'$
 $T' \rightarrow * F T' \mid \epsilon$
 $F \rightarrow (E) \mid \text{id}$

Add “error production”:
 $T' \rightarrow F T'$
 to ignore missing $*$, e.g.: **id id**

Pro:	Powerful recovery method
Cons:	Cannot be automated

	id	+	*	()	\$
E	$E \rightarrow T E'$			$E \rightarrow T E'$	<i>synch</i>	<i>synch</i>
E'		$E' \rightarrow + T E_R$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow F T'$	<i>synch</i>		$T \rightarrow F T'$	<i>synch</i>	<i>synch</i>
T'	$T' \rightarrow F T'$	$T' \rightarrow \epsilon$	$T' \rightarrow * F T'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$	<i>synch</i>	<i>synch</i>	$F \rightarrow (E)$	<i>synch</i>	<i>synch</i>

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?

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