Unit - II

CONTEXT FREE GRAMMARS AND PARSING

- Introduction
- Context-Free Grammars - Derivation, Parse trees, Ambiguity
- Types of Parsers
- LL(K) grammars and LL(1) parsing
- Bottom-up Parsing - handle pruning
- LR Grammar Parsing
- LALR parsing
- **Parsing ambiguous grammars**
- **Error Recovery in Parsing**
- YACC programming specification

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

Unit-II: Syntax Analysis (or) Parser

Using Ambiguous Grammars

Outline:

- Precedence and Associativity to Resolve Conflicts
- The “Dangling-Else” Ambiguity
- Error Recovery in LR parsing

Using Ambiguous Grammars

- Strictly speaking no **LALR parser** exists for **ambiguous grammar**.
- But, certain types of **ambiguous grammars** are quite useful in the specification and implementation of languages.
- For **large constructs** like expressions, an **ambiguous grammar** provides a shorter, more natural specification than any equivalent **unambiguous grammar**.

Using Ambiguous Grammars

- Even, for if-else construct, an **`ambiguous grammar`** provides more natural specification than its **`unambiguous grammar`**.
- Since, we are using **`ambiguous grammar`** to construct **`LR parsers`**, conflicts occur in the action part since there will be multiple entries in the parse table.

Using Ambiguous Grammars

The conflicts can be avoided as shown below:

- Using **`precedence and associativity`** to resolve the conflicts (In case of an expression)
- Avoiding **`dangling-else ambiguity`**

Precedence and Associativity to Resolve Conflicts

Example:

Obtain the **LR parsing table** for the following **ambiguous grammar**:

E′ → E$ where **$** indicates end of the input

1. E → E + E
2. E → E * E
3. E → (E)
4. E → id

Precedence and Associativity to Resolve Conflicts

Example: *Solution*

The **LR(0) items** for the above **augmented grammar** can be computed as in **SLR** and are shown below:

I_0:

E′ → .E

E → .E + E

E → .E * E

E → .(E)

E → .id

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

I_1: GOTO (I_0, E)

E′ → E.

E → E. + E

E → E. * E

I_2: GOTO (I_0, ()

E → (.E)

E → .E + E

E → .E * E

E → .(E)

E → .id

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

I_3: GOTO (I_0, id)

E → id.

I_4: GOTO (I_1, +)

E → E +.E

E → .E + E

E → .E * E

E → .(E)

E → .id

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

I_5: GOTO (I_1, *)

E → E *. E

E → .E + E

E → .E * E

E → .(E)

E → .id

I_6: GOTO (I_2, E)

E → (E.)

E → E. + E

E → E. * E

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

GOTO $(I_2, () = I_2$

E → (.E)

E → .E + E

E → .E * E

E → .(E)

E → .id

GOTO $(I_2, id) = I_3$

E → id.

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

I_7: GOTO (I_4, E)

E → E + E.

E → E. + E

E → E. * E

GOTO (I_4, () = I_2

E → (.E)

E → .E + E

E → .E * E

E → .(E)

E → .id

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

GOTO (I_4, id) = I_3

E → id.

I_8:GOTO (I_5, E)

E → E * E.

E → E. + E

E → E. * E

GOTO (I_5, () = I_2

GOTO (I_5, id) = I_3

I_9: GOTO (I_6,))

E → (E).

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

GOTO $(I_6, +) = I_4$

GOTO $(I_6, *) = I_5$

GOTO $(I_7, +) = I_4$

GOTO $(I_7, *) = I_5$

GOTO $(I_8, +) = I_4$

GOTO $(I_8, *) = I_5$

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

The **FIRST** and **FOLLOW** sets for the given grammar can be obtained as shown below:

	E
FIRST	(, id
FOLLOW	+,*,), $

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

Construction of SLR Parsing Table:

The parsing action function **ACTION** and **GOTO** can be obtained as shown below:

Precedence and Associativity to Resolve Conflicts

Example: *Solution*

The **ACTION** entries for terminals can be obtained as shown below:

Algorithm Rule 2.a

Transition GOTO $(I_i, a) = I_j$	ACTION [i, a] = shift j
I_0, (= I_2	[0, (] = s_2
I_0, id = I_3	[0, id] = s_3
I_1, + = I_4	[1, +] = s_4
I_1, * = I_5	[1, *] = s_5
I_2, (= I_2	[2, (] = s_2

Precedence and Associativity to Resolve Conflicts

Example: *Solution*

Construction of SLR Parsing Table:

Transition GOTO $(I_i, a) = I_j$	ACTION [i, a] = shift j
I_2, id = I_3	[2, id] = s_3
I_4, (= I_2	[4, (] = s_2
I_4, id = I_3	[4, id] = s_3
I_5, (= I_2	[5, (] = s_2
I_5, id = I_3	[5, id] = s_3

Precedence and Associativity to Resolve Conflicts

Example: *Solution*

Construction of SLR Parsing Table:

Transition GOTO $(I_i, a) = I_j$	ACTION [i, a] = shift j
I_6, + = I_4	[6, +] = s_4
I_6, * = I_5	[6, *] = s_5
I_6,) = I_9	[6,)] = s_9
I_7, + = I_4	[7, +] = s_4
I_7, * = I_5	[7, *] = s_5
I_8, + = I_4	[8, +] = s_4
I_8, * = I_5	[8, *] = s_5

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

The **ACTION** entries for the items ending with dot (.) are shown below: ***Algorithm Rule 2.b***

$[A \rightarrow \alpha.] \in I_i$	a = FOLLOW (A) then ACTION [i, a] = r $A \rightarrow \alpha$
$[E \rightarrow id.] \in I_3$	[3, {*, +,), $}] = r $E \rightarrow id$ (i.e., r_4) FOLLOW (E) = {*, +,), $}
$[E \rightarrow E + E.] \in I_7$	[7, {*, +,), $}] = r_1

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

$[A \rightarrow \alpha.] \epsilon I_i$	a = FOLLOW (A) then ACTION [i, a] = r $A \rightarrow \alpha$
$[E \rightarrow E * E.] \epsilon I_8$	[8, {*, +,), $}] = r_2
$[E \rightarrow (E).] \epsilon i_9$	[9, {*, +,), $}] = r_3

Precedence and Associativity to Resolve Conflicts

Example: *Solution*

$[S' \rightarrow S.] \epsilon I_i$ then **ACTION [i, $] = accept**: ***Algorithm Rule 2.c***

$[S' \rightarrow S.] \epsilon I_i$	ACTION [i, $] = accept
$[E' \rightarrow E.] \epsilon I_1$	[1, $] = accept

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

The **GOTO** states can be computed using *rule-3* are shown below: ***Algorithm Rule 3***

Transition GOTO $(I_i, A) = I_j$	Table GOTO [i, A] = j
I_0, E = I_1	[0, E] = 1
I_2, E = I_6	[2, E] = 6
I_4, E = I_7	[4, E] = 7
I_5, E = I_8	[5, E] = 8

Precedence and Associativity to Resolve Conflicts

Example: The final SLR parsing table:

	ACTION						GOTO
	id	+	*	(	)	$	E
0	S_3			S_2			1
1		S_4	S_5			acc	
2	S_3			S_2			6
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		S_4, r_1	S_5, r_1	r_1		r_1	
8		S_4, r_2	S_5, r_2	r_2		r_2	
9		r_3	r_3		r_3	r_3	
10							
11							

Precedence and Associativity to Resolve Conflicts

Example: ***Solution***

Since the grammar is ambiguous, it results in parsing-action conflicts when we produce the parsing table as shown above.

The sates corresponding to **I7** and **I8** generates these conflicts on input symbols **+** and *****.

Precedence and Associativity to Resolve Conflicts

How to avoid shift-reduce conflicts in a grammar that has arithmetic operators?

These conflicts can be resolved using precedence and associativity of operators as shown below:

1. If input operator and prefix on top of the stack to be reduced have same precedence and if the operator is left associative, then preference is given for reduction action.

Precedence and Associativity to Resolve Conflicts

How to avoid shift-reduce conflicts in a grammar that has arithmetic operators?

2. If input operator and prefix on top of the stack to be reduced have same precedence and if the operator is right associative, then preference is given for shift action.

Precedence and Associativity to Resolve Conflicts

How to avoid shift-reduce conflicts in a grammar that has arithmetic operators?

3. If input operator has less precedence than the operator present in the prefix that has to be reduced, then preference is given for reduce action.
4. If input operator has higher precedence than the operator present in the prefix that has to be reduced, then preference is given for shift action.

Precedence and Associativity to Resolve Conflicts

In our example

- Consider the entry: **action (7, +) = s_4 | r_1**. The conflict is whether to shift **4** or to reduce using **E → E + E** (since r_1 stands for reducer 1st production). When the input symbol is **+** and stack contains **E + E** and since operator **+** is left-associative, preference is given for reduction. So, retain r_1 and eliminate s_4.

Precedence and Associativity to Resolve Conflicts

In our example

- Consider the entry: **action (7, *) = s_5 | r_1**. The conflict is whether to shift **5** or to reduce using **E → E + E** (since r_1 stands for reducer 1st production). When the input symbol is ***** and stack contains **E + E** and since operator ***** has higher precedence than **+**, preference is given for shifting and not for reduction. So, retain s_5 and eliminate r_1.

Precedence and Associativity to Resolve Conflicts

In our example

- Consider the entry: **action (8, +) = s_4 | r_2**. The conflict is whether to shift **4** or to reduce using **E → E * E** (since r_2 stands for reducer 2nd production). When the input symbol is **+** and stack contains **E * E** and since operator ***** higher precedence than **+**, preference is given for reduction and not for shifting. So, retain r_2 and eliminate s_4.

Precedence and Associativity to Resolve Conflicts

In our example

- Consider the entry: **action (8, *) = s_5 | r_2**. The conflict is whether to shift **5** or to reduce using **E → E * E** (since r_2 stands for reducer 2nd production). When the input symbol is ***** and stack contains **E * E** and since operator ***** and stack contains **E * E** and since operator ***** is **left-associative,** preference is given for reduction. So, retain r_2 and eliminate s_5.

Precedence and Associativity to Resolve Conflicts

So, the final **parsing table** can be shown below:

	ACTION						GOTO
	id	+	*	(	)	$	E
0	S_3			S_2			1
1		S_4	S_5			acc	
2	S_3			S_2			6
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		r_1	S_5	r_1		r_1	
8		r_2	r_2	r_2		r_2	
9		r_3	r_3		r_3	r_3	
10							
11							

Avoiding dangling-else ambiguity

Example:

Obtain the **LR parsing table** for the following **ambiguous grammar**:

$S' \rightarrow S\$$ where **$** indicates end of the input

1. $S \rightarrow iSeS$
2. $S \rightarrow iS$
3. $S \rightarrow a$

Avoiding dangling-else ambiguity

Example: ***Solution***

The **LR(0) items** for the above **augmented grammar** can be computed as in **SLR** and are shown below:

I_0:

S′ → .S

S → .iSeS

S → .iS

S → .a

Avoiding dangling-else ambiguity

Example: ***Solution*** **LR(0) items**

I_1: GOTO (I_0, S)

S′ → S.

I_2: GOTO (I_0, i)

S → i.SeS

S → i.S

S → .iSeS

S → .iS

S → .a

Avoiding dangling-else ambiguity

Example: ***Solution*** **LR(0) items**

I_3: GOTO (I_0, a)

S → a.

I_4: GOTO (I_2, S)

S → iS.eS

S → iS.

Avoiding dangling-else ambiguity

Example: ***Solution*** **LR(0) items**

GOTO (I_2, i)= I_2

S → i.SeS

S → i.S

S → .iSeS

S → .iS

S → .a

GOTO (I_2, a)= I_3

S → a.

Avoiding dangling-else ambiguity

Example: ***Solution*** **LR(0) items**

GOTO (I_2, a)= I_3

S → a.

I_5:GOTO (I_4, e)

S → iSe.S

S → .iSeS

S → .iS

S → .a

Avoiding dangling-else ambiguity

Example: ***Solution*** **LR(0) items**

I_6: GOTO (I_5, S)

S → iSeS.

GOTO (I_5, i) = I_2

GOTO (I_5, a) = I_3

Avoiding dangling-else ambiguity

Example: ***Solution*** **FIRST** and **FOLLOW**

The **FIRST** and **FOLLOW** sets for the given grammar can be obtained as shown below:

	S
FIRST	i, a
FOLLOW	e, $

Avoiding dangling-else ambiguity

Example: *Solution* SLR Parsing table construction

	ACTION				GOTO
	i	e	a	$	S
0	s_2		s_3		1
1				acc	
2	s_2		s_3		4
3		r_3		r_3	
4		s_5, r_2		r_2	
5	s_2		s_3		6
6		r_1	r_1		

Avoiding dangling-else ambiguity

Example: ***Solution***

Observe that there are multiple entries in the above **LR parsing table**, since the given grammar is ambiguous. In the entry at **action (4, e) = s_5 | r_2** i.e., there is a conflict whether to shift **5** on to the stack or reduce using **2nd** production i.e., **S → iS**. If **S** is present on top of the stack instead of reducing, it is better to shift **5** which corresponds to **else**. This is because, the **else** is always associated with closest **if** and so instead of reducing, give preference for shifting. So, retain s_5 and eliminate r_2.

Avoiding dangling-else ambiguity

Example: ***Solution*** **Final SLR Parsing table is shown below:**

	ACTION				GOTO
	i	e	a	$	S
0	s_2		s_3		1
1				acc	
2	s_2		s_3		4
3		r_3		r_3	
4		s_5		r_2	
5	s_2		s_3		6
6		r_1	r_1		

Error Recovery in LR parsing

- An **LR parser** will detect an error when it consults the parsing action table and finds an error entry.
- Errors are never detected by consulting the goto table.
- An **LR parser** will announce an error as soon as there is no valid continuation for the portion of the input thus far scanned.
- A **canonical LR parser** will not make even a single reduction before announcing an error.

Error Recovery in LR parsing

- **SLR** and **LALR parsers** may make several reductions before announcing an error, but they will never shift an erroneous input symbol onto the stack.

In **LR parsing**, *we can implement* **panic-mode error recovery** *as follows.*

1. We scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. Zero or more input symbols are then discarded until a symbol **a** is found that can follow **A**.

Error Recovery in LR parsing

`Panic-mode error recovery:`

2. The parser then stacks the state **`GOTO(s, A)`** and resumes normal parsing. There might be more than one choice for the nonterminal **`A`**. Normally these would be nonterminals representing major program pieces, such as an expression, statement, or block. For example, if **`A`** is the nonterminal *stmt*, **`a`** might be **`semicolon`** or **`}`**, which marks the end of a statement sequence.

Error Recovery in LR parsing

Panic-mode error recovery:

3. This method of recovery attempts to eliminate the phrase containing the syntactic error. The parser determines that a string derivable from **A** contains an error. Part of that string has already been processed, and the result of this processing is a sequence of states on top of the stack.
4. The remainder of the string is still in the input, and the parser attempts to skip over the remainder of this string by looking for a symbol on the input that can follow **A**.

Error Recovery in LR parsing

`Panic-mode error recovery:`

5. By removing states from the stack, skipping over the input, and pushing **`GOTO(s, A)`** on the stack, the parser pretends that it has found an instance of **`A`** and resumes normal parsing.

Error Recovery in LR parsing

Phrase-level recovery:

1. Phrase-level recovery is implemented by examining each error entry in the **LR parsing table** and deciding on the basis of language usage the most likely programmer error that would give rise to that error.
2. An appropriate recovery procedure can then be constructed; evidently the top of the stack and/or first input symbols would be modified in a way deemed appropriate for each error entry.

Error Recovery in LR parsing

- In designing specific error-handling routines for an **LR parser**, we can fill in each blank entry in the action field with a pointer to an error routine that will take the appropriate action selected by the compiler designer.
- The actions may include insertion or deletion of symbols from the stack or the input or both, or alteration and transposition of input symbols.

Error Recovery in LR parsing

- We must make our choices so that the **`LR parser`** will not get into an infinite loop.
- A safe strategy will assure that at least one input symbol will be removed or shifted eventually, or that the stack will eventually shrink if the end of the input has been reached.
- Popping a stack state that covers a nonterminal should be avoided, because this modification eliminates from the stack a construct that has already been successfully parsed.

Error Recovery in LR parsing

Example:

Consider again the expression grammar:

1. **E → E + E**

2. **E → E * E**

3. **E → (E) | id**

Error Recovery in LR parsing

Example:

LR Parsing table for grammar is shown below:

	ACTION						GOTO
	id	+	*	(	)	$	E
0	S_3			S_2			1
1		S_4	S_5			acc	
2	S_3			S_2			6
3		r_4	r_4		r_4	r_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5		S_9		
7		r_1	S_5		r_1	r_1	
8		r_2	r_2		r_2	r_2	
9		r_3	r_3		r_3	r_3	

Error Recovery in LR parsing

Example:

The LR parsing table modified for error detection and recovery.

	ACTION						GOTO
	id	+	*	(	)	$	E
0	S_3	e_1	e_1	S_2	e_2	e_1	1
1	e_3	S_4	S_5	e_3	e_2	acc	
2	S_3	e_1	e_1	S_2	e_2	e_1	6
3	r_4	r_4	r_4	r_4	r_4	r_4	
4	S_3	e_1	e_1	S_2	e_2	e_1	7
5	S_3	e_1	e_1	S_2	e_2	e_1	8
6	e_3	S_4	S_5	e_3	S_9	e_4	
7	r_1	r_1	S_5	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	r_2	r_2	
9	r_3	r_3	r_3	r_3	r_3	r_3	

Error Recovery in LR parsing

Example: Error descriptions

e1: This routine is called from states **0**, **2**, **4** and **5**, all of which expect the beginning of an operand, either an **id** or a left parenthesis. Instead, **+**, *****, or the end of the input was found.

push state **3** (the goto of states **0**, **2**, **4** and **5** on **id**);

issue diagnostic "missing operand."

Error Recovery in LR parsing

Example: Error descriptions

e2: Called from states **0**, **1**, **2**, **4** and **5** on finding a right parenthesis.

remove the right parenthesis from the input;

issue diagnostic "unbalanced right parenthesis."

Error Recovery in LR parsing

Example: Error descriptions

e3: Called from states **1** or **6** when expecting an operator, and an **id** or right parenthesis is found.

push state **4** (corresponding to symbol **+**) onto the stack; issue diagnostic "missing operator."

Error Recovery in LR parsing

Example: Error descriptions

e4: Called from state **6** when the end of the input is found.

push state **9** (for a right parenthesis) onto the stack;
issue diagnostic “missing right parenthesis."

Summary...

Bottom-Up Parsing: **Using Ambiguous Grammars**

- Precedence and Associativity to Resolve Conflicts
- The "Dangling-Else" Ambiguity
- Error Recovery in LR parsing

Reading: Aho2, Section 4.8 (4.8.1, 4.8.2 & 4.8.3) & 4.6.5

Next Lecture: Parser Generators