THEORY OF COMPUTATION AND COMPILERS

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
The LR-Parsing Algorithm

Outline:

- Model of an LR parser
- LR-parsing algorithm
- Moves of an LR parser
- Example problem

Model of an LR Parser

A schematic of an **LR parser** is shown in Fig:

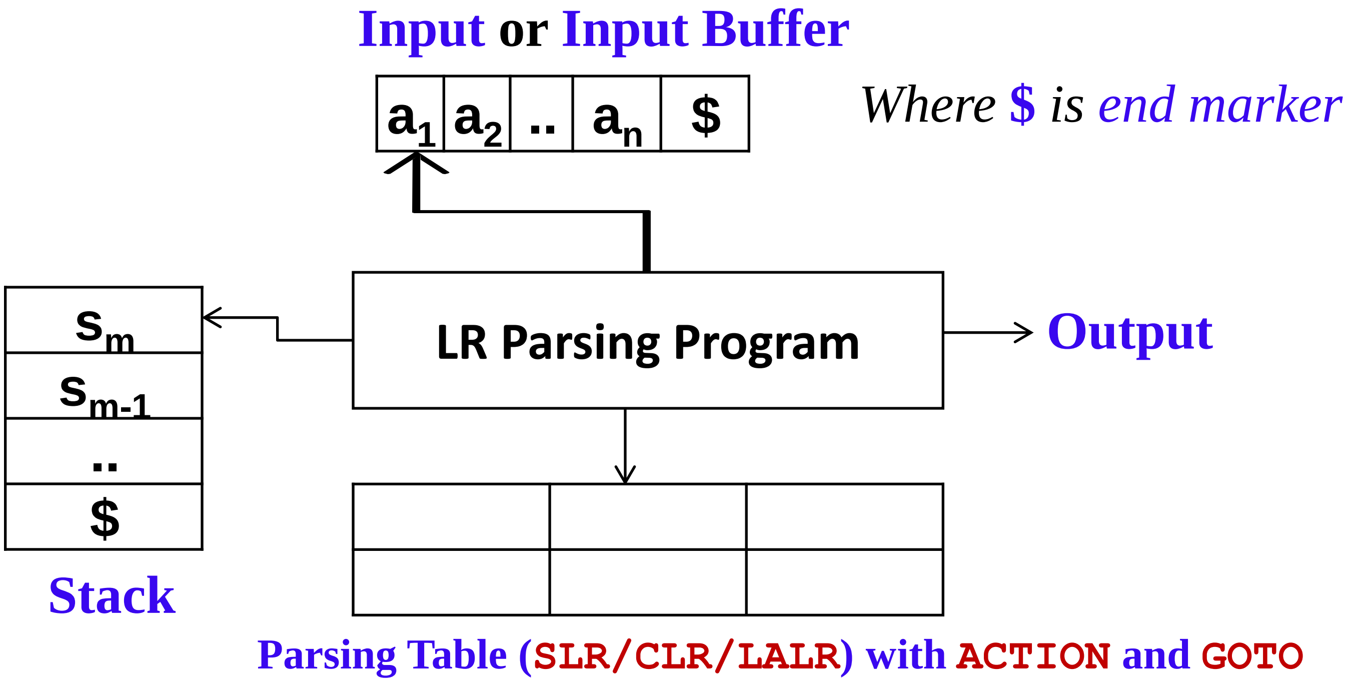

Figure: Model of an LR Parser

Model of an LR Parser

- **LR** parsers consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (**ACTION** and **GOTO**).
- The driver program is the same for all **LR** parsers; only the parsing table changes from one parser to another.
- The parsing program reads characters from an input buffer one at a time.
- Where a **shift-reduce** parser would shift a *symbol*, an **LR** parser shifts a *state*.

Model of an LR Parser

- Each *state* summarizes the information contained in the stack below it.
- The stack holds a *sequence of states*, $s_0 s_1 \ldots s_m$, where s_m is on top of the stack.
- In the **SLR** method, the stack holds states from the **LR(0)** automaton; the **Canonical LR (CLR)** and **LALR** methods are similar.
- By construction, each *state* has a corresponding grammar symbol.

Model of an LR Parser

- Recall that states correspond to sets of items, and that there is a transition from state i to state j if $\mathtt{GOTO}(I_i, X) = I_j$.
- All transitions to state j must be for the same grammar symbol X.
- Thus, each state, except the start state `0`, has a unique grammar symbol associated with it.

Model of an LR Parser

Structure of the LR Parsing Table:

The **parsing table** consists of two parts: a parsing-action function **ACTION** and a goto function **GOTO**.

1. The **ACTION** function takes as arguments a state ***i*** and a terminal ***a*** (or **$**, the input endmarker). The value of **ACTION[*i*, *a*]** can have one of four forms:

Model of an LR Parser

Structure of the LR Parsing Table:

a) **Shift** ***j***, where ***j*** is a state. The action taken by the parser effectively shifts input ***a*** to the stack, but uses state ***j*** to represent ***a***.

b) **Reduce** ***A*** $\rightarrow$ ***β***. The action of the parser effectively reduces ***β*** on the top of the stack to head ***A***.

c) **Accept**. The parser accepts the input and finishes parsing.

d) **Error**. The parser discovers an error in its input and takes some corrective action.

Model of an LR Parser

Structure of the LR Parsing Table:

2. We extend the **GOTO** function, defined on sets of items, to states: if $\mathbf{GOTO}[I_i, A] = I_j$, then **GOTO** also maps a state i and a nonterminal A to state j.

LR-Parser Configurations

To describe the behavior of an **LR parser**, it helps to have a notation representing the complete state of the parser: its stack and the remaining input.

A *configuration* of an **LR parser** is a pair:

$$(s_0 s_1 \ldots s_m,\ a_i a_{i+1} \ldots a_n \$)$$

where the first component is the stack contents (top on the right), and the second component is the remaining input.

LR-Parser Configurations

This configuration represents the right-sentential form

$$X_1X_2 \ldots X_m, \quad a_ia_{i+1} \ldots a_n$$

in essentially the same way as a shift-reduce parser would; the only difference is that instead of grammar symbols, the stack holds states from which grammar symbols can be recovered. That is, X_i is the grammar symbol represented by state s_i.

Note that s_0, the start state of the parser, does not represent a grammar symbol, and serves as a bottom-of-stack marker, as well as playing an important role in the parser.

Behavior of the LR Parser

The next move of the parser from the *configuration* $(s_0 s_1 \ldots s_m,\ a_i a_{i+1} \ldots a_n \$)$ is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and then consulting the entry $\mathbf{ACTION}[s_m,\ a_i]$ in the parsing action table.

Behavior of the LR Parser

The *configurations* resulting after each of the four types of move are as follows:

1. If **ACTION[s_m, a_i] = shift s**, then state s will be pushed on to the stack corresponding to the input symbol a_i , and the following configuration is obtained.

$$(s_0 s_1 \ldots s_m s,\ a_{i+1} \ldots a_n \$)$$

Behavior of the LR Parser

2. If $\texttt{ACTION}[s_m, a_i] =$ **reduce** $A \rightarrow \beta$, and if r is the length of β, remove r states from the stack and push s onto the stack where $s = \texttt{GOTO}[s_{m-r}, A]$ and the following configuration is obtained

$$(s_0 s_1 \ldots s_{m-r} s,\ a_i a_{i+1} \ldots a_n \$)$$

Behavior of the LR Parser

3. If **ACTION[s_m, a_i]= accept**, it indicates that parsing is successful.
4. If **ACTION[s_m, a_i]= blank**, then it is an error. The parser calls an error recovery routine.

Note: The initial configuration of the LR parser

Stack	Input
0	w$

where **0** is the **initial state** and **w** is the **input string**.

Behavior of the LR Parser

All **LR parsers** behave in this fashion; the only difference between one **LR parser** and another is the information in the **ACTION** and **GOTO** fields of the **parsing table**.

LR-parsing algorithm

Algorithm: **`LR-parsing`** algorithm

INPUT: An input string ***w*** and an **`LR-parsing table`** with functions **`ACTION`** and **`GOTO`** for a grammar **`G`**.

OUTPUT: If ***w*** is in ***L(G)***, the reduction steps of a **`bottom-up parser`** for ***w*** ; otherwise, an ***error*** indication.

LR-parsing algorithm

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program shown below:

```
let a be the first symbol of w$;
while(1) {/* repeat forever */
        let s be the state on top of the stack;
```

LR-parsing algorithm

METHOD:

```
if ( ACTION[s, a] = shift t ) {
        push t onto the stack;
        let a be the next input symbol;
} else if ( ACTION[s, a] = reduce A → β){
        pop |β|symbols off the stack;
        let state t now be on top of the stack;
        push GOTO[t, A] onto the stack;
        output the production A → β; }
```

LR-parsing algorithm

METHOD:

```
else if ( ACTION[s, a] = accept)
break; /* parsing is done */
else call error-recovery routine;
}
```

Moves of an LR parser

Example:

Show the sequence of moves made by the **LR parser** for the string **id + id * id** using the given grammar and the **LR parsing table**:

1. **E → E + T**
2. **E → T**
3. **T → T * F**
4. **T → F**
5. **F → (E)| id**
6. **F → id**

Moves of an LR parser

Example: **The given LR parsing table:**

	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				acc			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Moves of an LR parser

Example: ***Solution***

The sequence of moves made by the **LR parser** for the string **id + id * id** is shown below:

Stack	Input	Action
0	id+id*id$	S_5 => shift 5 onto the stack.
05	+id*id$	r_6 => Reduce using 6th production **F** → **id**

Note: The length of **id** on RHS of the production **F** → **id** is 1. So, remove one state (i.e., 5) from the stack and state 0 is on top of the stack. Now, see the GOTO table i.e., GOTO (0, F) = 3 in the table which is 3. Now, push 3 onto the stack.

Moves of an LR parser

Example: *Solution*

Stack	Input	Action
03	+id*id$	r_4 => Reduce using 4th production **T** → **F**. Pop |F| = 1 state from stack i.e., 3 and push GOTO (0, T) = 2 onto the stack.
02	+id*id$	r_2 => Reduce using 2nd production **E** → **T**. Pop |T| = 1 state from stack i.e., 2 and push GOTO (0, E) = 1 onto the stack.

Moves of an LR parser

Example: ***Solution***

Stack	Input	Action
01	+id*id$	s_6 => shift 6 onto the stack.
016	id*id$	s_5 => shift 5 onto the stack.
0165	*id$	r_6 => Reduce using 6th production $F \rightarrow id$. Pop |id| = 1 state from stack i.e., 5 and push GOTO (6, F) = 3 onto the stack.

Moves of an LR parser

Example: ***Solution***

Stack	Input	Action
0163	*id$	r_4 => Reduce using 4th production **T** → **F**. Pop |F| = 1 state from stack i.e., 3 and push GOTO (6, T) = 9 onto the stack.
0169	*id$	s_7 => shift **7** onto the stack.
01697	id$	s_5 => shift **5** onto the stack.

Moves of an LR parser

Example: ***Solution***

Stack	Input	Action
01697<u>5</u>	<u>$</u>	r_6 **=>** Reduce using 6th production **F** → **id**. Pop |id| = 1 state from stack i.e., 5 and push GOTO (7, F) = 10 onto the stack.
01697<u>10</u>	<u>$</u>	r_3 **=>** Reduce using 3rd production **T** → **T*F**. Pop |T*F| = 3 states from stack i.e., 10, 7 & 9 and push GOTO (6, T) = 9 onto the stack.

Moves of an LR parser

Example: ***Solution***

Stack	Input	Action
0169710	$	r_3 => Reduce using 3rd production **T** → **T*F**. Pop |T*F| = 3 states from stack i.e., 10, 7 & 9 and push GOTO (6, T) = 9 onto the stack.
0169	$	r_1 => Reduce using 1st production **E** → **E+T**. Pop |E+T| = 3 states from stack i.e., 9, 6 & 1 and push GOTO (0, E) = 1 onto the stack.

Moves of an LR parser

Example: ***Solution***

Stack	Input	Action
016<u>9</u>	<u>$</u>	r_1 => Reduce using 1st production **E** → **E+T**. Pop |E+T| = 3 states from stack i.e., 9, 6 & 1 and push GOTO (0, E) = 1 onto the stack.
0<u>1</u>	<u>$</u>	**ACCEPT**, Parsing is successful.

Note: If **ACTION** [s_m, a_i] = **blank**, then it is <u>an error</u> and <u>parsing is not successful</u>.

Summary...

Bottom-Up Parsing: LR-parsing algorithm

- Model of an LR parser
- LR-parsing algorithm
- Moves of an LR parser
- Example problem

Reading: Aho2, Section 4.6.3

Next Lecture: Simple LR parser (SLR parser)