Compiler Design ReferencePoint Suite

Table of Contents

Point 1: Understanding LR Parsers

Rajesh Kaushal

A parser is a program that accepts a sequence of tokens from the source program and generates a parse tree. Tokens are the symbols used in the programming language. A parser uses grammar to construct a parse tree for the expressions of a source program.

A grammar for a programming language is similar to English grammar. A grammar gives a precise and easy-to-understand structure to a programming language. A grammar for a programming language consists of production rules, which are used to construct statements in the programs.

An LR parser is used to parse the tokens of the source language. Parsing involves gathering the tokens and arranging them in some ordered structure such as tree or tables. The L in LR stands for left-to-right and the R stands for rightmost derivation in reverse.

This ReferencePoint discusses LR parsers and the construction of Simple LR (SLR), Look Ahead LR (LALR), and Canonical LR (CLR) parsing tables. It also explains how to use ambiguous grammars. It discusses automatic parser generation, implementation of an LR parsing table, and constructing LALR sets of items.

Basics of Parsing

A parser takes the tokens of a source program as input and arranges them into a hierarchical order called a parse tree. The compiler then uses the parse tree to generate the code for the source program. Figure 2-1-1 shows the complete compilation process:

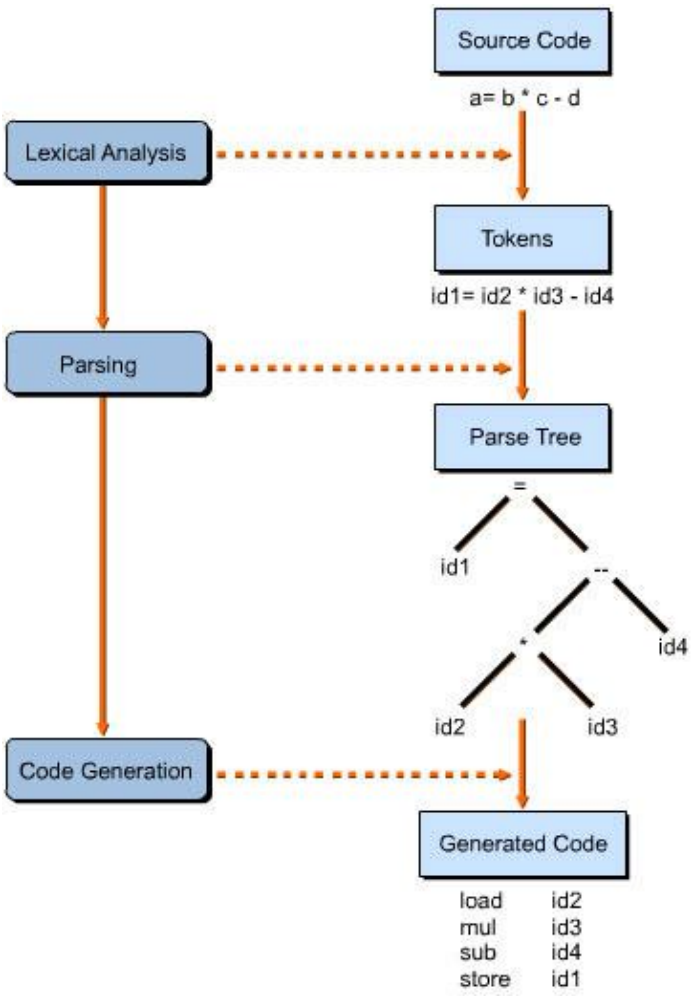

Figure 2-1-1: The Compilation Process
The compiler operates in three main phases: lexical analysis, parsing, and code generation. The source code fragment is an expression, a = b * c – d. After the lexical analysis phase, the expression, a = b * c – d, is translated into the sequence of tokens, id1 = id2 * id3 – id4. The parser parses the sequence of tokens and constructs a parse tree for the sequence of tokens. The code generation phase of a compiler generates the code from the parse tree.

A compiler takes the source code as input from the end user. The lexical analysis phase of the compiler produces a sequence of tokens for this source code. The parser then takes the sequence of tokens from the lexical analyzer and generates a parse tree for the given sequence of tokens. The code analyzer generates executable code for the source code.

Types of LR Parsers

Parsers use grammars to parse tokens. Grammars consist of production rules, which are used by the parser to match the tokens. There are two basic techniques of parsing:

- Top-down: Starts parsing with the left-most token from the sequence of tokens. Constructs a parse tree for the tokens starting from the root and creates the leaf-nodes of the parse tree in preorder.
- Bottom-up: Starts parsing with the right-most token from the sequence of tokens. Constructs a parse tree for the tokens starting from the leaf-nodes of the parse tree and then constructs the root node.

An LR parser is based on the bottom-up parsing technique. There are three types of LR parsers:

- SLR: Produces a parsing tree for a limited set of grammars.
- CLR: Produces a parsing table for all types of grammars.
- LALR: Produces a parse tree for programming language grammar constructs.

Note A parsing table is a tabular representation of a parse tree. A compiler uses a parsing table to generate the binary code for the source program.

Finite Automata

A finite automaton (FA) is a mathematical model of a computer system. It defines the number of internal states that a computer system has to traverse starting from the input state to the output state. The number of internal states is different for different types of computations. For example, the internal states to add two numbers will be different from that to multiply the numbers. An FA consists of:

- A finite set of states of the computer system.
- A finite set of transitions from the input state to the output state.

A state indicates the internal configuration for the computer for each input symbol. For example, when a computer reads the word, while, which is a C language keyword, the computer has five

distinct states. When the computer reads the first character, w, it is in the state one. When it reads the character, h, the computer is in state two. The computer changes its state after reading each character of the keyword.

A transition diagram is associated with every FA that shows a state change. Figure 2-1-2 shows the transition diagram to read the while keyword:

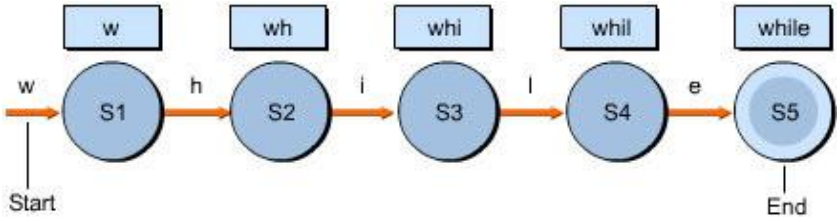

Figure 2-1-2: Transition Diagram

There are two approaches to calculate the output of the FA. When the output is associated with the state, then the FA is called Moore machine. When the output is associated with the transition, then the FA is called Mealy machine.

Transition diagrams are represented as tables, called transition tables. These tables are based on FA and are used to construct the parse tables.

There are two types of FAs:

- Deterministic FA (DFA): There is only one state for each input state.
- Nondeterministic FA (NFA): Multiple states are possible for each input symbol.

The input to a computer is in the form of strings, which are a collection of letters of the alphabet, such as a, b, c, x, z, and symbols, such as arithmetic operators. Input strings are drawn from any formal language. A formal language is a set of finite-length strings or words over some finite alphabet. In computer science, C, Fortran, and Java, are examples of a formal language.

A formal language is constructed from formal grammars. A formal grammar consists of a finite set of terminal symbols, a finite set of nonterminal symbols, a set of production rules, and a start symbol. A terminal symbol cannot be expanded or divided into smaller units. A nonterminal symbol can be expanded or divided further into sub units. The MIT linguistics professor Noam Chomsky defined a hierarchy of classes of formal grammars that generate formal languages. Table 2-1-1 describes the four types of formal grammars:

Table 2-1-1: Types of Formal Grammar

Grammar	Language	Automaton	Production Rules
Type-0	Recursively enumerable	Turing machine	No restrictions, Permits productions of the form , where and are arbitrary strings of grammar symbols.
Type-1	Context-sensitive	Linear-bounded nondeterministic Turing machine	Permits productions of the form , where length of and are same.
Type-2	Context-free	Nondeterministic pushdown automaton	Permits productions of the form , where consists of single nonterminal symbol.
Type-3	Regular	Finite state automaton	Permits productions of the form , where consists of a single nonterminal symbol and the length of is finite.

Context-Free Grammar

Context-free grammar (CFG) specifies how to write production rules for any language. Production rules identify tokens in a source program and break the source program into a sequence of tokens. The identified tokens are arranged in a table called a parse table. Compilers also use tokens to generate executable code for the source program.

Parsers use CFG to specify the syntax of a source program. CFG describes the hierarchical structure of a programming language. Understanding CFG requires familiarity with the following CFG components:

- Tokens: Include lexical units of the programming language. For example, tokens in C include the keywords if, else, while, for, the parentheses and the semicolon, any variable or expression, and the operators used in a program.
- Terminal symbols: Include a set of tokens that cannot be subdivided into smaller units. For example, in C, you cannot subdivide the if or else keywords into smaller lexical units.
- Nonterminal symbols: Include a set of tokens that can be divided into smaller lexical units. For example, you can divide the statement, a+b, into a, b, and the + operator.
- Production rules: Include the rules used to divide nonterminal token into smaller lexical units.

For any programming language, the CFG consists of:

- A set of terminal tokens.
- A set of nonterminal tokens.
- A set of production rules where each rule is written in the following manner:

  Nonterminal token Sequence of terminal tokens or nonterminal tokens

- Designation of a nonterminal token as the start symbol.

You define a statement of a programming language using CFG components. For example, the form of an if-else statement in C is:

```
if (condition)
statement;
else
statement;
```

The if-else statement consists of the if keyword, an opening parenthesis, an expression for a condition, a closing parenthesis, a statement, the else keyword, and another statement. For example, if the expr and stmt variables represent the condition and the statement, respectively, then the if-else statement can be expressed as:

```
stmt   if (expr) stmt else stmt
```

In the above statement, the arrow means: can have the form. The statement, stmt if (expr) stmt else stmt, is called a production rule. The, if and else keywords and the parentheses are the lexical units of C. These lexical units are called tokens. The expr and stmt variables represent sequences of tokens and are called nonterminals.

The production rule, stmt if (expr) stmt else stmt, represents a generalized if-else statement in C. You can use this rule to identify tokens used in the if-else statement.

Derivation and Reduction

A grammar for a programming language defines a set of rules that determines how the language is constituted. For example, you can categorize a sentence in the English language as an ordered sequence of nouns, verbs, adjectives, or prepositions. The sentence itself is based on a fixed formats defined by the English grammar. Similarly, a grammar for a programming language defines rules to construct valid and correct programming statements. This set of rules is called production rules.

To obtain a valid expression or a statement of the language, production rules are applied to the grammar symbol. Examples of grammar symbols are letters of the alphabet, digits, arithmetic operators, and punctuation marks. You replace a symbol using one of the production rules and then rewrite the symbol. After a series of replacements, you obtain a valid expression that consists of tokens.

In reduction, you replace a symbol by another symbol according to a grammar production. You use this technique to derive the expression and statements of a language using grammar rules. There are two types of derivations:

- Left-most: Starts with applying a production rule to the left-most nonterminal symbol of the expression.
- Right-most: Starts with applying a production rule to the right-most nonterminal symbol of the expression.

The following grammar rules help understand the left- and right-most derivations:

S aAS

S a

A SbA

A SS

A ba

The string aabbaa is derived using production rules. The same string is derived using left-most and right-most derivations.

Using the left-most derivation:

1. Start with the first rule:

   S aAS

2. Apply production rule three, A SbA, in the expression, S aAS, and replace A with SbA. The expression now is:

   S aSbAS

3. Apply production rule two, S a, in the expression, S aSbAS, and replace S with a. The expression now is:

   S aabAS

4. Apply production rule five, A ba, in the expression, S aabAS, and replace A with ba. The expression now is:

   S aabbaS

5. Apply production rule two, S a, in the expression, S aabbaS, and replace S with a. The expression now is:

   S aabbaa

The above procedure derives the final expression, aabbaa, from the production rules of grammar using left-most derivation.

Using right-most derivation:

1. Start with the first rule, S aAS.

   S aAS

2. Apply production rule two, S a, in the expression, S aAS, and replace S with a. The expression now is:

   S aAa

3. Apply production rule three, A SbA, in the expression, S aAa, and replace A with SbA, The expression now is:

   S aSbAa

4. Apply production rule five, A ba, in the expression, S aSbAa, and replace A with ba. The expression now is:

   S aSbbaa

5. Apply production rule two, S a, in the expression, S aSbbaa, and replace S with a. The expression now is:

   S aabbaa

The above procedure derives the final expression, aabbaa, from the production rules of grammar using right-most derivation.

Parse Trees

A parse tree diagrammatically describes how CFG derives an expression from the programming language. A parse tree has the following properties:

- The root node of the tree is represented by the start symbol of the CFG production rule.
- Each leaf node of the tree can either be represented by a terminal token or be empty.
- Each interior node of the tree can only be represented by a nonterminal.

The concept of a parse tree can be understood by drawing a tree for the derivation of the expression, 5 + 7 * 2.

There are four production rules to recognize the syntax of the expression 5 + 7 * 2:

- expr expr * digit
- expr expr + digit
- expr digit
- digit 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

In the above grammar, the first three production rules start with the nonterminal, expr.

The parser can recognize the expression, 5 + 7 * 2, and all other similar expressions, such as 1 + 2 * 3 or 9 + 8 * 3. The expression, 5 + 7 * 2, can be derived by applying the four production rules of the grammar:

1. Start with the first production rule, which is written as:

   ```
   expr   expr * digit
   ```

2. Replace the expr expression using the second production rule, expr expr + digit. The derived expression is:

   ```
   expr   expr + digit * digit
   ```

3. Replace the expr expression using the third production rule, expr digit. The derived expression is:

   ```
   expr   digit + digit * digit
   ```

4. Apply the fourth production rule, digit 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9, to the above expression to derive the original expression, 5 + 7 * 2. Similarly, you can match all expressions based on the pattern, digit + digit * digit.

Figure 2-1-3 shows the parse tree for the expression, 5 + 7 * 2:

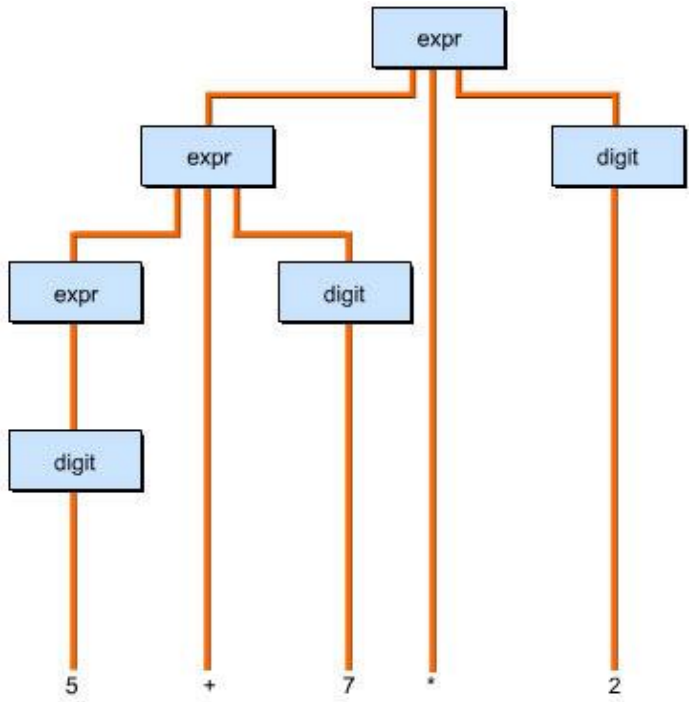

Figure 2-1-3: The Parse Tree for 5 + 7 * 2
In the above figure, the top level node, expr is written as expr * digit. After this, the symbol, expr in expr * digit is expanded and written as expr + digit * digit. Next, the expression, expr + digit * digit is written as digit + digit * digit. Finally, the expression, digit + digit * digit is written as, 5 + 7 * 2, which is the original expression.

Ambiguous Grammar

A CFG that produces multiple parse trees for an expression is called ambiguous. Ambiguity occurs when multiple interpretations are derived from a single sentence.

If an expression can be represented by two parse trees, the CFG of that expression is ambiguous. Some ambiguous grammars are useful in the specification of languages. The following example shows the ambiguity in the expression, a + b * c. The production rules for the grammar are:

```
expr   expr * expr
expr   expr + expr
expr   id
id   x | y | z
```

You can draw a parse tree for the expression, x + y * z, in two ways.

Figure 2-1-4 shows the first parse tree for the expression, x + y * z:

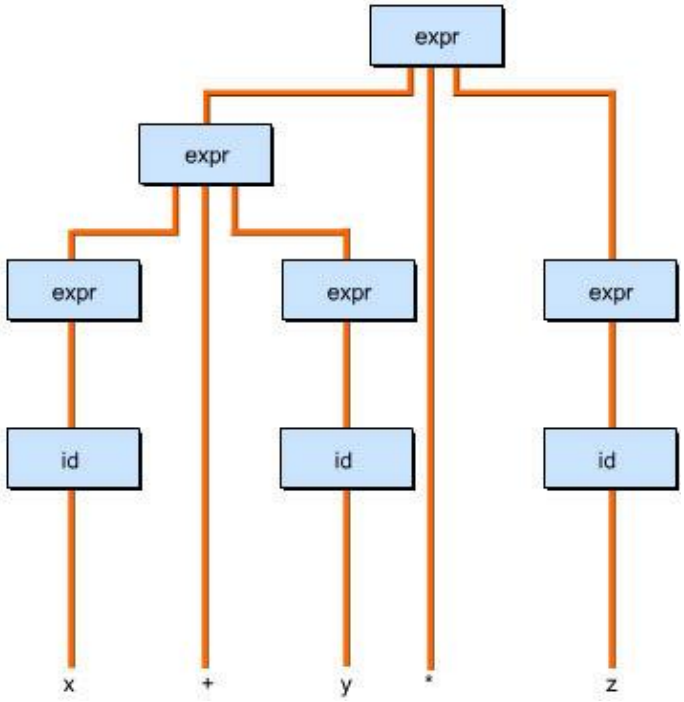

Figure 2-1-4: First Parse Tree for x + y * z

Figure 2-1-5 shows another parse tree for the same expression, x+ y * z:

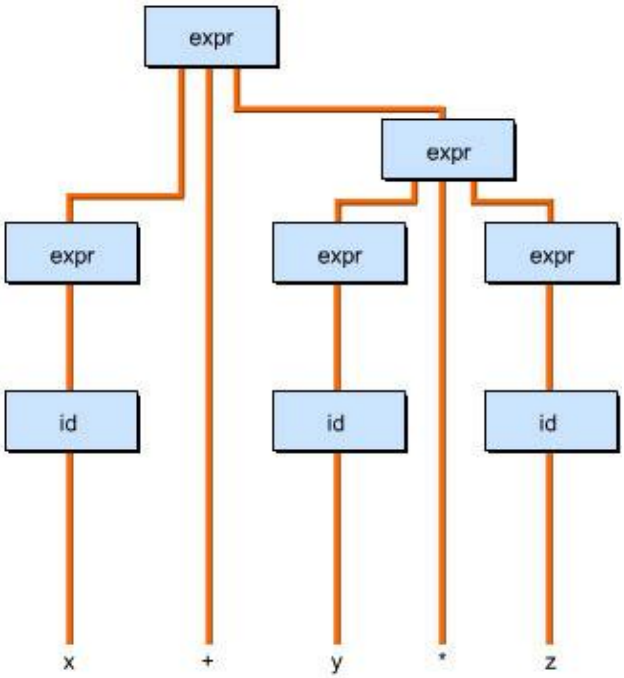

Figure 2-1-5: Second Parse Tree for x + y * z

LR Parsing

An LR parser is an efficient shift/reduce parser used to parse a large class of grammars. A shift/reduce parser constructs a parse tree for an input string that begins at the bottom of the tree and constructs the root node. LR parsers provide several advantages, including recognizing CFG constructs and detecting syntactic errors.

An LR parser generator locates the ambiguities and constructs of the grammar that are difficult to parse.

LR parsing involves four steps:

- Shift: Takes one token from the sequence of tokens and shifts it to the stack.
- Reduce: Finds a production rule from the given grammar that matches the token and reduces the token.
- Accept: If the token matches the production rule of the grammar exactly, then the parser accepts the token and takes a new token from the unparsed tokens.
- Error: Generates an error if the token does not match the grammar.

The above steps consist of shifting and reducing tokens, a process that explains why LR parsing is also called shift/reduce parsing.

Architecture of an LR Parser

An LR parser consists of an input buffer, an output buffer, a stack, an LR program routine, and a parsing table. An LR parser reads the input in the form of a group of tokens. It reads the input from left to right, one token at a time. The terminal and nonterminal symbols in each state are stored in the Action and goto parsing tables, respectively.

A parsing table lists all the actions taken by a parser. Each parsing table consists of three tables:

- State: Consists of different states of a parser.
- Action: Consists of the action functions for a parser.
- Goto: Consists of the goto functions for a parser.

Figure 2-1-6 shows an LR parser:

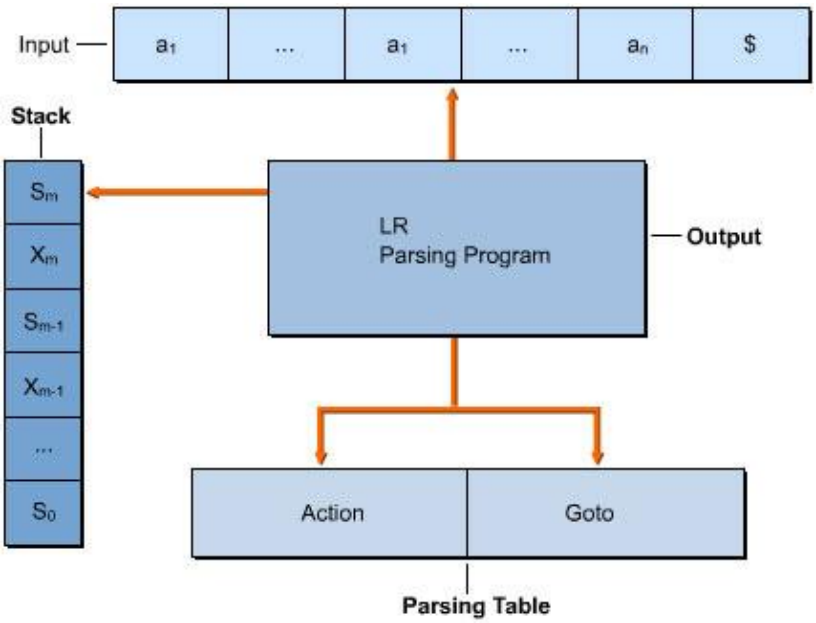

Figure 2-1-6: Diagrammatic Representation of an LR Parser

In Figure 2-1-6, the parser reads the input from the input buffer one token at a time. The parser uses a stack to store a string of the form $S_0X_1 S_1X_2 ...X_mS_m$, where S_m is on the top of the stack, X_i refers to a grammar symbol, and S_i represents the state of the parser.

The LR parser uses grammar to construct a parsing table in which every entry is uniquely defined. To parse a sequence of strings, an LR parser:

1. Takes the input token from the input buffer.
2. Pushes the token onto the stack.
3. Checks the parsing table for related action for the token on the stack.
4. Performs the action on the token.
5. Repeats steps 1 through 4.

You can adapt the LR parsing technique to parse ambiguous grammars. For example, when you construct an LR parser for a grammar that contains two productions, S iCtS | iCtSeS, there is a state point where a parsing action conflicts with e, either shifting or reducing the grammar tokens by S iCtS. If the conflict is resolved in favor of shifting, then the construction of the LR parser continues.

LR Parsing Algorithm

To understand the LR parsing algorithm, you need to understand how an LR parser works. An LR parser uses a data structure called configuration to store the sequence of parsed and unparsed tokens. A configuration is a pair of LR parser stack components and a sequence of unparsed tokens, as in:

```
(S0 X1 S1 X2 S2    ...   Xm Sm , ai ai+1   ...   an $ )
```

In the above example:

- S_m represents the top of the stack.
- The symbols, a_i a_{i+1} ... a_n, are unparsed tokens and are stored in the input buffer.
- The symbol, a_i, is the first symbol to be processed.
- The symbols, S_0 X_1 S_1 X_2 S_2 ... X_m S_m, are stored on the stack and S_m is the top of the stack.

The algorithm determines the next move of the parser by reading the current input symbol a_i and the top of the stack S_m. The algorithm starts parsing with the current state from the top of the stack and then the next token from the input buffer. The start state is state zero. The action for state zero and the token from the input buffer is defined in the parsing table. The parser checks the parsing table for action on the token. After performing the action on the token, the parser makes the entries in the configuration. To parse a sequence of tokens, the parser performs the following actions on the sequence:

- If Action [S_m, a_i] =shift, then the LR parser makes a shift move and enters the configuration in the following stack:

  $(S_0\ X_1\ S_1X_2\ S_2...S_m\ X_m,\ a_i\ a_{i+1}....a_n\ \$)$

  The parser then shifts the current input token, a_i, and the next state, S=goto[S_m, a_i], onto the stack. The current input symbol changes to a_{i+1}.

- If Action [S_m, a_i]=reduce A , then the LR parser makes a reduce move and enters the following string:

  $(S_0\ X_1\ S_1X_2\ S_2...S_m\ X_m,\ a_i\ a_{i+1}....a_n\ \$)$

  Here, S=goto [S_{m-r}, a] and r store the length of the grammar. The LR parser pops the r stack symbols and the r grammar symbols from the stack and pushes A and p on the stack. The current input token remains unchanged during the reduce move.

- If Action[S_m, a_i] = accept, then the parsing process is complete.
- If Action[S_m, a_i] = error, the error is identified and the parser calls the error recovery routine.

The initial configuration state of the LR parser is S_0, a_1 a_2 ... a_n $, where S_0 is the initial state and a_1 a_2 ... a_n is the sequence of unparsed tokens. The parser then performs the necessary actions on the sequence of unparsed tokens to generate a parse tree.

Listing 2-1-1 shows the algorithm for an LR parser:

Listing 2-1-1: Algorithm for an LR Parser

```
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), a bottom-up parse for W, otherwise an error indication
```

```
Method: Initially, the parser has S0 is the initial state and W$ in the input buffer.
The parser then executes the program until an accept or error action is encountered.
Variables: Integer ip;
Set ip to point to the first symbol of W$.
Repeat forever begin
        Let, S be the state on top of the stack and
       A the symbol pointed to by ip;
if action[S, A] = shift Sx on top of the stack;
        Push A and Sx on top of the stack;
        Advance ip to the next input symbol
end
else if action[S, A] = reduce A     then begin
        Pop 2*|   | symbols off the stack;
        Let Sx be the state now on the top of the stack;
        Push A then goto[Sx , A] on the top of the stack;
        Output the production A
end
else if action[S, A] = accept then
        Return
else error()
end
```

The above listing shows the algorithm for an LR parser.

The grammar to accept the binary arithmetic operators, + and * is:

```
E ? E + T
E ? T
T ? T * F
T ? F
F ? (E)
F ? id
```

Table 2-1-2 lists the LR parsing table for the above grammar to accept the binary arithmetic operators, + and *:

Table 2-1-2: LR Parsing Table

State	Action						goto		
-	Id	+	*	(	)	$	C	B	A
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	-	-	-

In Table 2-1-2, S_i is the shift action for the stack state, i. The symbol, r_j, is the reduce action for the state number represented by j. The symbol, acc, denotes that an accept action is encountered

Table 2-1-3 lists the moves by the parser on the input, id * id +id, with the sequence of the stack and contents of the input for the grammar:

Table 2-1-3: Parsing of the Input id * id +id

Stack	Input	Action
0	id * id + id$	Shift
0 id 5	* id + id$	Reduce by F id
0 F 3	* id + id$	Reduce by T F
0 T 2	* id + id$	Shift
0 T 2 * 7	* id + id$	Shift
0 T 2 * 7 id 5	+ id$	Reduce by F id
0 T 2 * 7 B 10	+ id$	Reduce by T T*F
0 T 2	+ id$	Reduce by E id
0 E 1	+ id$	Shift
0 E 1 + 6	id$	Shift
0 E 1 + 6 id 5	$	Reduce by F id
0 E 1 + 6 F 3	$	Reduce by T F
0 E 1 + 6 T 9	$	E E + T
0 E 1	$	Accept

Initially, the LR parser is in state 0 and encounters id as the first input symbol. The LR parser checks the parsing table for the id symbol entry in state 0. The corresponding action is S_5, which shifts the id symbol and changes the state from 0 to 5. The LR parser checks each input and the state of the stack and takes necessary actions according to the parsing table.

Constructing Parsing Tables

Although all LR parsers, such as SLR, LALR, and CLR, use the same parsing algorithm, they use different parsing tables. For example, the SLR parser uses the SLR parsing table. Similarly, the LALR and CLR parsers use the LALR and CLR parsing tables.

Constructing an LR Parsing Table

LR parsing tables are implemented for a grammar with 50 to 100 terminals and 100 productions. A grammar for which you can construct an LR parsing table is called LR grammar.

The following is an example of regular grammar:

```
E ? C + A
E ?A
A ?A * B
A ?B
B ? (E)
B ? id
```

Table 2-1-4 lists the LR parsing table for the above grammar:

Table 2-1-4: Parsing Table

State	Action						goto		
-	Id	+	*	(	)	$	E	B	A
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	-	-	-

In the above table, input symbols are sequence of tokens, such as id, +, *, (,), and $. The numbers 0 through 11 represent the different states of the DFA, which accepts the input symbols. The action table contains entries such as S_5, R_1, or S_{11}. These entries define the actions taken by the parser on the input tokens.

There are different actions defined for each input token. Table 2-1-5 lists the representation of the actions for the tokens id, any, and (:

Table 2-1-5: Actions Common in an LR Parsing Table

Symbol	Action
Id	S_5
(	S_4
Any	Error

Table 2-1-5 above shows that action for the token, id is S_5. This means shift action for the token, id.

Each state defines actions for some input tokens. For example, state 1 defines actions for the input tokens + and $.

Table 2-1-6 lists the actions in state 1:

Table 2-1-6: Actions in State 1

Symbol	Action
+	S_6
$	Acc
Any	Error

In state 2, error entries can be replaced by r_2. As a result, reduction by production 2 occurs on any input except on *. Table 2-1-7 lists the actions for state 2:

Table 2-1-7: Actions for State 2

Symbol	Action
*	S_7
Any	R_2

State 3 has only error and r_4 entries. You can define actions for the states 5, 10, and 11. Table 2-1-8 lists the actions for state 8:

Table 2-1-8: Actions for State 8

Symbol	Action
+	S_6
)	S_{11}
Any	Error

State 9 defines the action, S_7, for the input token, *. For any other input token, the action defined is R_1. Table 2-1-9 lists the actions for state 9:

Table 2-1-9: Actions for State 9

Symbol	Action
*	S_7
Any	R_1

You can implement a goto table using list data structure. A list for each nonterminal, A, is implemented as a pair of current-state and next-state, which indicates:

```
goto [current-state, A] = next- state.
```

This technique is useful because any one column of the goto table has very few states. The goto state on a nonterminal A can only be the state derived from a set of items in which some items have A immediately to the left of a dot. No set has items with X and Y immediately to the left of a dot if X is not equal to Y.

Error entries in the goto table are never consulted by the parser. To further reduce space usage, these error entries can be replaced by the most correct entry in the column.

Note To learn more about errors and how to correct them, refer to the Error Detection and Recovery ReferencePoint.

Constructing an SLR Parsing Table

In the SLR parsing table, the parsing action and goto functions are constructed from the DFA that recognizes the possible prefixes. Although the DFA does not produce uniquely defined parsing action tables for all grammars, it constructs a parsing table on multiple programming language grammars.

To construct an SLR parsing table, the algorithm takes an augmented grammar as input. The following is a definition of augmented grammar: If G is a grammar with start symbol S, then G', which is the augmented grammar for G, is G with a new start symbol S' and production S' S. Listing 2-1-2 shows the algorithm to construct an SLR parsing table:

Listing 2-1-2: Algorithm for an SLR Parsing Table

```
Input: An augmented grammar G'
Output: The SLR parsing table functions, action and goto for G'.
Method:
Begin,
Construct, C = {P0, P1, ..., Pi,.... Pn} is the collection of sets of LR (0) items for the augme
 The states of the parser are P0, P1, 2...Pn. The parsing actions for the state i are:
If [A    $ b  ] in Pi and the goto (Pi, b) = Pj, then set action (i, b) to shift j.
If [A     $] in Pi, then set action (i, b) to reduce A  .
If [S'   S $] in Pi, then set action (i, $) to accept.
If goto (Pi, A) = Pj, then set goto [i, A] = j.
End
```

Any entry that is not defined by the rules is set to error. If any entry has more than one action, then the grammar is not called SLR.

For example, the grammar to construct the parsing action and goto function is:

```
C ? C + A
C ?A
A ?A * B
A ?B
B?(C)
B ? id
```

In this algorithm, the parsing action and the goto states can be set in the parsing table.

Table 2-1-10 shows the SLR parsing table for the above grammar:

Table 2-1-10: SLR Parsing Table for a Grammar

State	Action						goto		
-	Id	+	*	(	)	$	C	B	A
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	-	-	-

A shift/reduce conflict is a parser state when the parser cannot decide whether to perform the shift or perform reduce actions. This situation arises when the parser has multiple production rules for reductions. The following grammar shows that a shift/reduce conflict can occur while an SLR parsing table is being constructed:

```
S   L=R
S   R
L   *R
L   id
R   L
I0:
                  S  . L=R
                  S  . R
                  R  . L
                  L  . *R
                  L  . Id
I1:
                  S   L.= R (1)
                  R   L.(2)
```

The above grammar shows shift/reduce conflicts in an LR parser when the parser performs reduction using the first rule.

Another common conflict is the reduce and reduce conflict, that can also occur while an SLR parsing table is being constructed. For example, in the following production rules for A and B:

```
If A    and B
```

Choosing the production for on the stack becomes difficult because both productions for A and B give the same value, .

An SLR parsing table has three limitations:

- It has numerous reduce actions.
- It focuses on the reduce move in the second state.
- It considers two states of DFA to recognize the viable prefixes.

Constructing an LALR Parsing Table

You construct an LALR (1) parsing table from the configuration sets in the input state token. The number 1 in the parenthesis means that the LALR parser takes one input symbol of look ahead at a time. Similarly, LR (1) means that the LR parser takes one input symbol of look ahead at a time. In the absence of merging states in the configuration sets, the LALR (1) table is similar to the corresponding LR (1) table.

If an LALR table contains states that can be merged, it has fewer rows than an LR table. For a typical programming language grammar construct, an LR table can have several thousand rows. The number of rows is substantially reduced because of merging for LALR.

Although the LALR table is similar to the SLR and LR (0) tables, which have the same number of states because of merging, LALR has fewer reduce actions. The number 0 in the parenthesis means that the parser does have any look ahead input symbol. You use the brute force method to construct an LALR parsing table. This method involves constructing an LR (1) parsing table and merging its sets. Listing 2-1-3 shows the algorithm to construct an LALR parsing table:

Listing 2-1-3: Algorithm to Construct an LALR Parsing Table

```
Input: An augmented grammar G'.
Output: The LALR parsing table functions, action and goto for G'.
Method:
Begin,
Construct C = { I0, I1, ... In},
 the collection of sets of LR (1) states from the given grammar.
For each core present in the set of LR(1) items, find all sets having that core,
and replace these sets by their union.
Let C' = {J0, j1, ..., Jm} be the resulting sets of LR(1) items. The parsing actions for state I
 Ji. If there is a parsing action conflict,  the algorithm fails to
 produce a parser, and the grammar is not LALR.
The parsing table is constructed as follows: if J is the union of one or more sets of LR items,
that is, J = I1⋃ I2⋃ ... ⋃Ik, then the cores of goto (I1, X),
goto (I2, X), ... goto (Ik, X) are the same, since I1, I2, ... Ik all have the same core.
Let K be the union of all sets of items with the same core as goto (I1, X).
Then goto (j, X) = K.
End
```

For example, you can use the following grammar to construct an LR parsing table:

```
S' ?S
S ?CC
C? cC I d
```

Table 2-1-11 lists the LR table for the above grammar:

Table 2-1-11: LR Parsing Table

State	Action			goto	
-	C	D	$	S	C
0	S_3	S_4	-	1	2
1	-	-	acc	-	-
2	S_6	S_7	-	-	5
3	S_3	S_4	-	-	8
4	R_3	r_3	-	-	-
5	-	-	r_1	-	-
6	S_6	S_7	-	-	9
7	-	-	r_3	-	-
8	R_2	r_2	-	-	-
9	-	-	r_2	-	-

The LR parsing table uses the grammar, S' ?S, S ?CC, and C? cC I d. An LR table can contain states that are similar and have the same core. These states are merged into one state to form the LALR parsing table from the LALR grammar.

Next, you construct the LALR parsing table for the above grammar, as shown in Table 2-1-12:

Table 2-1-12: LALR Parsing Table

State	Action			goto	
-	C	D	$	S	C
0	S_{36}	S_{47}	-	1	2
1	-	-	Acc	-	-
2	S_{36}	S_{47}	-	-	5
36	S_{36}	S_{47}	-	-	89
47	r_3	r_3	r_3	-	-
5	-	-	r_1	-	-
89	r_2	r_2	r_2	-	9

Constructing a CLR Parsing Table

In the SLR method, the state i calls for reduction by A >·on the input symbol, a, if the set of items, I_i, contains [A >.] and a is in FOLLOW (A). In situations when the state i appears on top of the stack, a possible prefix,(,)·, may be on the stack such that, A cannot be succeeded by a in a right sentential form. The reduction by A > would be invalid on A. Carrying more information in the state enables you to rule out some of the invalid reductions by A >. Listing 2-1-4 shows the algorithm for a canonical LR parsing table:

Listing 2-1-4: Algorithm for a CLR Parsing Table

```
Input: An augmented grammar G'.
Output: An augmented LR parsing table functions, action and goto for G'.
Method:
Begin,
Construct C = {P0, P1, ...Pi, ... Pn},
the collection of sets of LR(1) items for G. Pi gives the state, i.
The parsing actions for state I are determined as follows:
If [A   .A , b] is in Pi and goto (Pi, a) = Pj, then set action (i, a) to "shift j".
If [A    ., a] in Pi, then set action (i, a) to reduce A  .
If [S'   S., $] in Pi, then set action (i, $) to accept
End
```

The above algorithm fails in the following cases:

- The goto transition state for i, for the following condition:

```
If goto (Ii, A) = Ij, then goto [i, A] = j.
```

- The set containing item [S'?.S,$] gives rise to the initial state of the parser.

If conflicts arise, then the grammar is not LR (1) and the algorithm does not produce the parsing table. The table formed from the parsing action and goto functions with this algorithm is called the canonical LR (1) parsing table.

For example, the following is a sample grammar:

```
S' ?S
S ?CC
C? cC I d
```

Table 2-1-13 lists the Canonical LR table for the above grammar:

Table 2-1-13: Canonical LR Table

State	Action			goto	
-	C	D	$	S	C
0	S_3	S_4	-	1	2
1	-	-	acc	-	-
2	S_6	S_7	-	-	5
3	S_3	S_4	-	-	8
4	r_3	r_3	-	-	-
5	-	-	r_1	-	-
6	S_6	S_7	-	-	9
7	-	-	r_3	-	-
8	r_2	r_2	-	-	-
9	-	-	r_2	-	-

Related Topics

For related information on this topic, you can refer to:

- Introducing Compiler Design
- Principles of Compiler Design
- Error Detection and Recovery