Introduction:

In computer science, LR parsers are a type of bottom-up parsers that efficiently handle deterministic context-free languages in guaranteed linear time. The LALR parsers and the SLR parsers are common variants of LR parsers. LR parsers are often mechanically generated from a formal grammar for the language by a parser generator tool. They are very widely used for the processing of computer languages, more than other kinds of generated parsers.

- The name LR is an initialism. The L means that the parser reads input text in one direction without backing up; that direction is typically Left to right within each line, and top to bottom across the lines of the full input file. (This is true for most parsers.)
- The R means that the parser produces a reversed Rightmost derivation; it does a bottom-up parse, not a top-down LL parse or ad-hoc parse.
- The name LR is often followed by a numeric qualifier, as in LR(1) or sometimes LR (*k*). To avoid backtracking or guessing, the LR parser is allowed to peek ahead at *k* lookahead input symbols before deciding how to parse earlier symbols. Typically *k* is 1 and is not mentioned.
- The name LR is often preceded by other qualifiers, as in SLR and LALR.
- LR parsers are deterministic; they produce a single correct parse without guesswork or backtracking, in linear time. This is ideal for computer languages. But LR parsers are not suited for human languages which need more flexible but slower methods.
- Other parser methods (CYK algorithm, Earley parser, and GLR parser) that backtrack or yield multiple parses may take O(n^2), O(n^3) or even exponential time when they guess badly.
- The above properties of L, R, and k are actually shared by all shift-reduce parsers, including precedence parsers. But by convention, the LR name stands for the form of parsing invented by Donald Knuth, and excludes the earlier, less powerful precedence methods (for example Operator-precedence parser).
- LR parsers can handle a larger range of languages and grammars than precedence parsers or top-down LL parsing. This is because the LR parser waits until it has seen an entire instance of some grammar pattern before committing to what it has found.
- An LL parser has to decide or guess what it is seeing much sooner, when it has only seen the leftmost input symbol of that pattern. LR is also better at error reporting. It detects syntax errors as early in the input stream as possible.

Example:

- An LR parser scans and parses the input text in one forward pass over the text. The parser builds up the parse tree incrementally, bottom up, and left to right, without guessing or backtracking.
- At every point in this pass, the parser has accumulated a list of subtrees or phrases of the input text that have been already parsed.
- Those subtrees are not yet joined together because the parser has not yet reached the right end of the syntax pattern that will combine them.
- At step 6 in the example parse, only "A*2" has been parsed, incompletely. Only the shaded lower-left corner of the parse tree exists. None of the parse tree nodes numbered 7 and above exist yet.
- Nodes 3, 4, and 6 are the roots of isolated subtrees for variable A, operator *, and number 2, respectively. These three root nodes are temporarily held in a parse stack. The remaining unparsed portion of the input stream is "+ 1". (Please refer the below image)

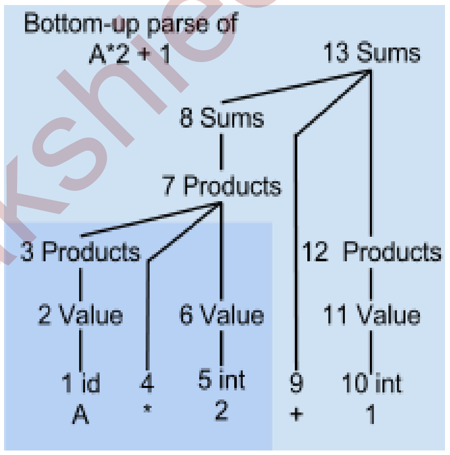
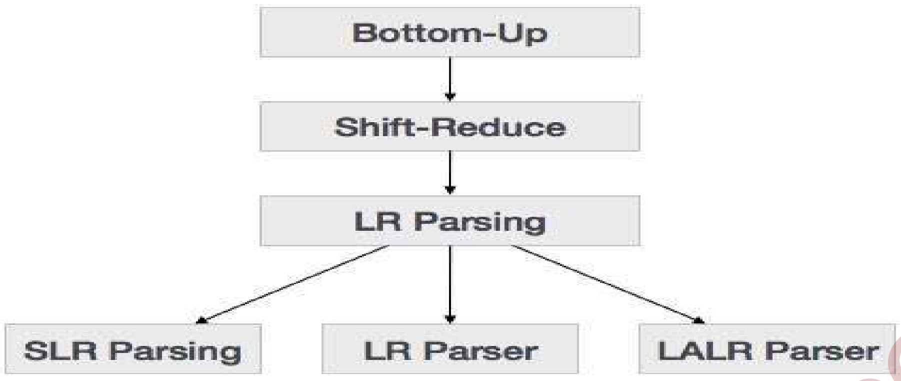


Diagram for Bottom-Up Parsing:



Shift and reduce actions:

As with other shift-reduce parsers, an LR parser works by doing some combination of Shift steps and Reduce steps.

- A **Shift** step advances in the input stream by one symbol. That shifted symbol becomes a new single-node parse tree.
- A **Reduce** step applies a completed grammar rule to some of the recent parse trees, joining them together as one tree with a new root symbol.

If the input has no syntax errors, the parser continues with these steps until all of the input has been consumed and all of the parse trees have been reduced to a single tree representing an entire legal input.

LR parsers differ from other shift-reduce parsers in how they decide when to reduce, and how to pick between rules with similar endings. But the final decisions and the sequence of shift or reduce steps are the same. Much of the LR parser's efficiency is from being deterministic. To avoid guessing, the LR parser often looks ahead (rightwards) at the next scanned symbol, before deciding what to do with previously scanned symbols. The lexical scanner works one or more symbols ahead of the parser. The **look ahead** symbols are the 'right-hand context' for the parsing decision.

Example:

- At every parse step, the entire input text is divided into parse stack, current lookahead symbol, and remaining unscanned text.
- The parser's next action is determined by the rightmost stack symbol(s) and the lookahead symbol.

- The action is read from a table containing all syntactically valid combinations of stack and lookahead symbols.

Step	Parse Stack	Look Ahead	Unscanned	Parser Action
0	*Empty*	*id*	= B + C*2	Shift
1	*Id*	=	B + C*2	Shift
2	*id* =	*id*	+ C*2	Shift
3	*id* = *id*	+	C*2	Reduce by Value ← *id*
4	*id* = Value	+	C*2	Reduce by Products ← Value
5	*id* = Products	+	C*2	Reduce by Sums ← Products
6	*id* = Sums	+	C*2	Shift
7	*id* = Sums +	*id*	*2	Shift
8	*id* = Sums + *id*	*	2	Reduce by Value ← *id*
9	*id* = Sums + Value	*	2	Reduce by Products ← Value
10	*id* = Sums + Products	*	2	Shift
11	*id* = Sums + Products *	*int*	*Eof*	Shift
12	*id* = Sums + Products * *int*	*eof*		Reduce by Value ← *int*
13	*id* = Sums + Products * Value	*eof*		Reduce by Products ← Products * Value
14	*id* = Sums + Products	*eof*		Reduce by Sums ← Sums + Products
15	*id* = Sums	*eof*		Reduce by Assign ← *id* = Sums
16	Assign	*eof*		Done

Grammar Examples:

A grammar is the set of patterns or syntax rules for the input language. It doesn't cover all language rules, such as the size of numbers, or the consistent use of names and their definitions in the context of the whole program. Shift-reduce parsers use a context-free grammar that deals just with local patterns of symbols.

The example grammar used here is a tiny subset of the Java or C language:

Assign ← *id* = Sums

Sums ← Sums + Products

Sums ← Products

Products ← Products * Value

Products ← Value

Value ← *int*

Value ← *id*

- The grammar's terminal symbols are the multi-character symbols or 'tokens' found in the input stream by a lexical scanner. Here these include **= + *** and *int* for any integer constant, and *id* for any identifier name.
- The grammar doesn't care what the *int* values or *id* spellings are, nor does it care about blanks or line breaks. The grammar uses these terminal symbols but does not define them. They are always at the bottom bushy end of the parse tree.
- The capitalized terms like Sums are nonterminal symbols. These are names for concepts or patterns in the language. They are defined in the grammar and never occur themselves in the input stream. They are always above the bottom of the parse tree.
- They only happen as a result of the parser applying some grammar rule. Some nonterminals are defined with two or more rules; these are alternative patterns. Rules can refer back to themselves.
- This grammar uses recursive rules to handle repeated math operators. Grammars for complete languages use recursive rules to handle lists, parenthesized expressions and nested statements.
- Any given computer language can be described by several different grammars. The grammar for a shift-reduce parser must be unambiguous itself, or be augmented by tie-breaking precedence rules.
- This means there is only one correct way to apply the grammar to a given legal example of the language, resulting in a unique parse tree and a unique sequence of shift/reduce actions for that example.

- A table-driven parser has all of its knowledge about the grammar encoded into unchanging data called parser tables. The parser's program code is a simple generic loop that applies unchanged to many grammars and languages.
- The tables may be worked out by hand for precedence methods. For LR methods, the complex tables are mechanically derived from a grammar by some parser generator tool like Bison.
- The parser tables are usually much larger than the grammar. In other parsers that are not table-driven, such as recursive descent, each language construct is parsed by a different subroutine, specialized to the syntax of that one const.

LR and LALR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- SLR(1) – Simple LR Parser:
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction
- LR(1) – LR Parser:
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- LALR(1) – Look-Ahead LR Parser:
  - Works on intermediate size of grammar
  - Number of states are same as in SLR(1)

LR Parsing Algorithm

Here we describe a skeleton algorithm of an LR parser:

```
token = next_token()
```

```
repeat forever
  s = top of stack

  if action[s, token] = “shift si” then
    PUSH token
    PUSH si
    token = next_token()

  else if action[s, tpken] = “reduce A::= β“ then
    POP 2 * |β| symbols
    s = top of stack
    PUSH A
    PUSH goto[s,A]

  else if action[s, token] = “accept” then
    return

  else
    error()
```

LL vs. LR

LL	**LR**
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.

Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Error Recovery in Parsing:

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic Mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement Mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error Productions

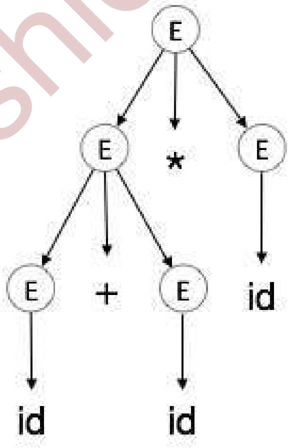
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

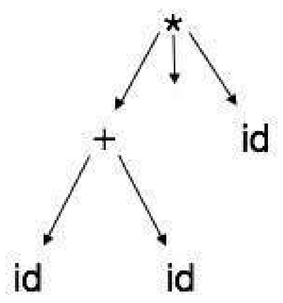
The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Abstract Syntax Trees

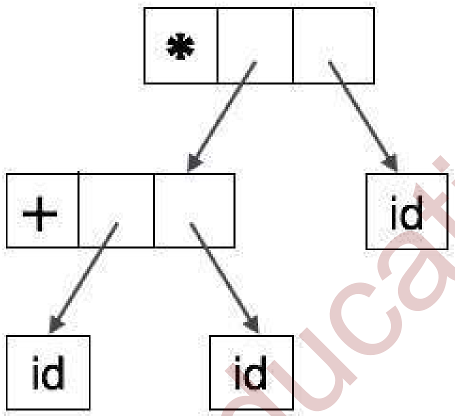
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



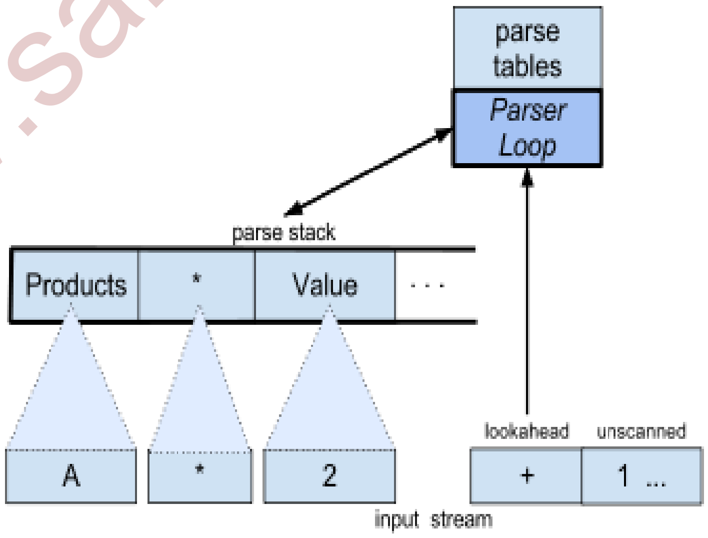
If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.



YACC:

- YACC is a computer program for the Unix operating system. It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of thesource code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF.
- YACC itself used to be available as the default parser generator on most Unix systems, though it has since been supplanted as the default by more recent, largely compatible, programs.
- YACC is an acronym for "Yet Another Compiler Compiler". It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of thesource code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF.
- It was originally developed in the early 1970s by Stephen C. Johnson at AT&T Corporation and written in the B programming language, but soon rewritten in C. It appeared as part of Version 3 Unix and a full description of Yacc was published in 1975.
- The input to Yacc is a grammar with snippets of C code (called "actions") attached to its rules. Its output is a shift-reduce parser in C that executes the C snippets associated with each rule as soon as the rule is recognized.
- Typical actions involve the construction of parse trees. Using an example from Johnson, if the call node(label, left, right)constructs a binary parse tree node with the specified label and children, then the rule recognizes summation expressions and constructs nodes for them. The special identifiers $$, $1 and $3 refer to items on the parser's stack.

```
expr : expr '+' expr  { $$ = node('+', $1, $3); }
```

- Yacc and similar programs (largely reimplementations) have been very popular. Yacc itself used to be available as the default parser generator on most Unix systems, though it has since been supplanted as the default by more recent, largely compatible, programs such as Berkeley Yacc, GNU bison, MKS Yacc and Abraxas PCYACC.

- An updated version of the original AT&T version is included as part of Sun's OpenSolaris project. Each offers slight improvements and additional features over the original Yacc, but the concept and syntax have remained the same.
- Yacc has also been rewritten for other languages, including OCaml, Ratfor, ML, Ada, Pascal, Java, Python, Ruby,Go and Common Lisp.
- Yacc produces only a parser (phrase analyzer); for full syntactic analysis this requires an external lexical analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper.
- Lexical analyzer generators, such as Lex or Flex are widely available. The IEEE POSIX P1003.2 standard defines the functionality and requirements for both Lex and Yacc.
- Some versions of AT&T Yacc have become open source. For example, source code (for different implementations) is available with the standard distributions of Plan 9 andOpenSolaris.