for implementing syntax analyzers for programming languages work on subclasses of unambiguous grammars and have complexity $O(n)$.

A recursive-descent parser is an LL parser that is implemented by writing code directly from the grammar of the source language. EBNF is ideal as the basis for recursive-descent parsers. A recursive-descent parser has a subprogram for each nonterminal in the grammar. The code for a given grammar rule is simple if the rule has a single RHS. The RHS is examined left to right. For each nonterminal, the code calls the associated subprogram for that nonterminal, which parses whatever the nonterminal generates. For each terminal, the code compares the terminal with the next token of input. If they match, the code simply calls the lexical analyzer to get the next token. If they do not, the subprogram reports a syntax error. If a rule has more than one RHS, the subprogram must first determine which RHS it should parse. It must be possible to make this determination on the basis of the next token of input.

Two distinct grammar characteristics prevent the construction of a recursive-descent parser based on the grammar. One of these is left recursion. Although we do not cover it, an algorithm exists to remove both direct and indirect left recursion from a grammar. The other problem is detected with the pairwise disjointness test, which tests whether a parsing subprogram can determine which RHS is being parsed on the basis of the next token of input. Some grammars that fail the pairwise disjointness test can be modified to pass it, using left factoring.

The parsing problem for bottom-up parsers is to find the substring of the current sentential form that must be reduced to its associated LHS to get the next (previous) sentential form in the rightmost derivation. This substring is called the handle of the sentential form. A parse tree can provide an intuitive basis for recognizing a handle. A bottom-up parser is a shift-reduce algorithm, because in most cases it either shifts the next lexeme of input onto the parse stack or reduces the handle that is on top of the stack.

The LR family of shift-reduce parsers is the most commonly used bottom-up parsing approach for programming languages, because parsers in this family have several advantages over alternatives. An LR parser uses a parse stack, which contains grammar symbols and state symbols to maintain the state of the parser. The top symbol on the parse stack is always a state symbol that represents all of the information in the parse stack that is relevant to the parsing process. LR parsers use two parsing tables, ACTION and GOTO. The ACTION part specifies what the parser should do, given the state symbol on top of the parse stack and the next token of input. The GOTO table is used to determine which state symbol should be placed on the parse stack after a reduction has been done.
4. Write and test the code to implement the state diagram of Problem 2.
5. For those rules that pass the test in Problem 1, write a recursive-descent parsing subprogram that parses the language generated by the rules. Assume you have a lexical analyzer named lex and an error-handling subprogram named error, which is called whenever a syntax error is detected.
6. For those rules that pass the test in Problem 2, write a recursive-descent parsing subprogram that parses the language generated by the rules. Assume you have a lexical analyzer named lex and an error-handling subprogram named error, which is called whenever a syntax error is detected.
7. Implement and test the LR parsing algorithm given in Section 4.5.3.