UNIT-1

Evolution of programming languages:

- Years 50: Creation of high-level languages (closer to humans).

- Years 60: Expansion of specialized languages. Forth. Simula I. Lisp, Cobol. Trying unsuccessfully to impose general languages: Algol, PL / 1.

- Years 70: Duel between structured programming with Pascal and efficiency of C language. Basic generalized on personal computers from 1977, until the late 80s.

- Years 80: Experimentating other ways including objects. ML. Smalltalk. On computers, we now use C, Pascal, Basic compiled.

- Years 90: Generalization of object-oriented programming with the performance of microcomputers. Java, Perl, Python languages in addition to microphones.

- 2000s: Internet Programming (and future innovations, see end of text).

- Years 2010: Concurrency and asynchronicity. JavaScript and Go languages among others help to create online fluid applications.

Fortran - 1954-1958 - FORmula TRANslator system

John Backus and other researchers at IBM

Language dedicated to mathematical calculations.
Fortran II (1958) introduced SUB-ROUTINES, FUNCTIONS, LOOPS, a primitive FOR control structure. Identifiers were limited to six characters.

Attribute grammars - 1965

Donald Knuth

Completing the BNF method, attribute grammars describes the semantic of languages to be made of functions. This type of grammar helps the building of compilers.

UNIT-1

Fortran - 1954-1958 - FORmula TRANslator system

John Backus and other researchers at IBM

Language dedicated to mathematical calculations.

UNCOL - 1958 - Universal Computer Oriented Language

This is the first bytecode, the first intermediate language, addressing the need to be able to write programs that run on all hardware

Lisp - 1958-1960 - List Processing

Mac Carthy

Functional language for list processing.
It is purely recursive, and not iterative. There is no difference between code and data.

COBOL - 1960 - Common Business Oriented Language.

Cobol is a classical procedural language aimed at enterprise management, in which a program is divided in 4 divisions: identification, environment, data, procedure.

Smalltalk - 1972

This is a fully object oriented language wich runs always inside a graphical environment, with windows, mouse, etc...

2. Describing Syntax:

A language, whether natural (such as English) or artificial (such as Java), is a set of strings of characters from some alphabet. The strings of a language are called **sentences** or statements.

(or)

The syntax of a language is set of rules that defines the form of a languages . they define low expressions,sentences,statements and program units are formed by the fundamental units called word/lexemes .

- The syntax rules of a language specify which strings of characters from the language's alphabet are in the language
- The lexemes of a programming language include its numeric literals, operators, and special words, among others. One can think of programs as strings of lexemes.

- Token is a description of the lexems.
  Example: `index = 2 * count + 17;`

The lexemes and tokens of this statement are

Lexemes	tokens
Index	identifier
=	equal_sing
2	int_literal
*	mult_op
Count	identifier
+	plus_op
17	int_literal
;	semicolon

2.1. Language Recognizers:

In general, languages can be formally defined in two distinct ways: by **recognition** and by **generation** (although neither provides a definition that is practical by itself for people trying to learn or use a programming language).

A language recognizer is part of the compiler, responsible for syntax analysis of the translating language.

- Suppose we have a language 'L'
- That uses an alphabet' ∑ 'of characters.

To define L formally using the recognition method, we would need to construct a mechanism R, called a recognition device, capable of reading strings of characters from the alphabet ∑.

- R would indicate whether a given input string was or was not in L.

R would either

- accept or reject the given string.
- If R, when fed any string of characters over∑, accepts it only if it is in L, then R is a description of L.
- The syntax analysis part of a compiler is a recognizer for the language the compiler translates.
- the recognizer need not test all possible strings of characters from some set to determine whether each is in the language.
- the syntax analyzer determines whether the given programs are syntactically correct. The structure of syntax analyzers, also known as parsers.

Example: let 'L' be a language that consists of a string of characters to define a language using the recognition method , a mechanism called automata is constructed

1. An automata is a recognition from 'L' the automata is designed in such a way that it can read the strings and determines.
2. Whether a particular string is a part of the language or not .
3. The automata then accepts or rejects.

2.2. Language Generators:

A language generator is a device that can be used to generate the sentences of a language. We can think of the generator as having a button that produces a sentence of the language every time it is pushed.

- By contrast, the syntax-checkingportion of a compiler (a language recognizer) is not as useful a language description for a programmer because it can be used only in trial-and-error mode.
- To determine the correct syntax of a particular statement using a compiler.
- the programmer can only submit a speculated version and note whether the compiler accepts it On the other hand.
- It is often possible to determine whether the syntax of a particular statement is correct by comparing it with the structure of the generator.

Backus-Naur Form and Context-Free Grammars:

In the middle to late 1950s, two men, Noam Chomsky and John Backus, in unrelated research efforts, developed the same syntax description formalism, which subsequently became the most widely used method for programming language syntax.

Backus-Naur Form:

The ACM-GAMM group began designing ALGOL 58. A landmark paper describing ALGOL 58 was presented by John Backus, a prominent member of the ACM-GAMM group, at an international conference in 1959 (Backus, 1959).

- The ACM-GAMM group began designing ALGOL 58. A landmark paper describing ALGOL 58 was presented by John Backus, a prominent member of the ACM-GAMM group, at an international conference in 1959 (Backus, 1959).
- ALGOL 60 (Naur, 1960). This revised method of syntax description became known as **Backus-Naur Form**, or simply **BNF.**
- It is remarkable that BNF is nearly identical to Chomsky's generative devices for context-free languages, called **context-free grammars**. In the remainder of the chapter, we refer to context-free grammars.

3.Context-Free Grammars:

In the mid-1950s, Chomsky, a noted linguist (among other things), described four classes of generative devices or grammars that define four classes of languages (Chomsky, 1956, 1959).

- The forms of the tokens of programming languages can be described by regular grammars.

- The syntax of whole programming languages, with minor exceptions, can be described by context-free grammars. Because Chomsky was a linguist, his primary interest was the theoretical nature of natural languages.
- V□(VUT)* and V,T,P,S
- V is a non terminals.
- T is a terminals.
- P is a productions.
- S is a starting productions.

3.1 Fundamentals:

A **metalanguage** is a language that is used to describe another language. BNF is a metalanguage for programming languages.

A simple Java assignment statement, for example, might be represented by the abstraction <assign> (pointed brackets are often used to delimit names of abstractions). The actual definition of <assign> can be given by

Example:

<assign> → <var> = <expression>

- The text on the left side of the arrow, which is aptly called the **left-hand side** (LHS), is the abstraction being defined.
- The text to the right of the arrow is the definition of the LHS. It is called the **right-hand side** (RHS).
- And consists of some mixture of tokens, lexemes, and references to other abstractions.
- Altogether, the definition is called a**rule**, or **production**. In the example rule just given, the abstractions <var>and <expression> obviously must be defined for the <assign> definition to be useful.
- Multiple definitions can be written as a single rule, with the different definitions separated by the symbol|, meaning logical OR. For example, a Java **if** statement can be described with the rules.
- <if_stmt> → **if** (<logic_expr>) <stmt>
  <if_stmt> → **if** (<logic_expr>) <stmt> **else** <stmt>

  OR with the rule

  <if_stmt> → **if** (<logic_expr>) <stmt>
  | **if** (<logic_expr>) <stmt> **else** <stmt>

3.2Describing Lists:

The describing lists of syntactic elements in programming languages (for example, a list of identifiers appearing on a data declaration statement).

For BNF, the alternative is recursion. A rule is **recursive** if its LHS appears in its RHS. The following rules illustrate how recursion is used to describe lists:

Ex:

<ident_list> → identifier

| identifier, <ident_list>

This defines <ident_list> as either a single token (identifier) or an identifier followed by a comma and another instance of <ident_list>.

3.3Grammars and Derivations:

A grammar is a generative device for defining languages. The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the **start symbol**. This sequence of rule applications is called a **derivation.**

The start symbol represents a complete program and is often named <program>.

A program consists of the special word **begin**, followed by a list of statements separated by semicolons, followed by the special word **end**.

A Grammar for a Small Language

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
| <var> – <var>
| <var>

A derivation of a program in this language follows:

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

this case <program>. The symbol => is read “derives.” Each successive string in the sequence is derived from the previous string by replacing one of the nonterminals with one of that nonterminal’s definitions.

3.4Parse Trees:

One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the languages they define. These hierarchical structures are called **parse trees.**

For example, the parse tree .
A = B * (A + C)

Every internal node of a parse tree is labeled with a nonterminal symbol; every leaf is labeled with a terminal symbol. Every subtree of a parse tree describes one instance of an abstraction in the sentence.

3.5 Ambiguity:

A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be **ambiguous.**

Ambiguous Grammar for Simple Assignment Statements:

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

The grammar of Example 3.5 is ambiguous because the sentence A = B + C * A

Has two distinct parse trees, The ambiguity occurs because the grammar specifies slightly less syntactic structure than does the grammar of Rather than allowing the parse tree of an expression to grow only on the right, this grammar allows growth on both the left and the right.

Syntactic ambiguity of language structures is a problem because compilers often base the semantics of those structures on their syntactic form. Specifically, the compiler chooses the code to be generated for a statement by examining its parse tree.

3.6 Extended BNF:

Because of a few minor inconveniences in BNF, it has been extended in several ways. Most extended versions are called Extended BNF, or simply EBNF, even though they are not all exactly the same.

The extensions do not enhance the descriptive power of BNF; they only increase its readability and writability.

For example:

C **if-else** statement can be described as

<if_stmt> → **if** (<expression>) <statement> [**else** <statement>]

Without the use of the brackets, the syntactic description of this statement would require the following two rules:

<if_stmt> → **if** (<expression>) <statement>
| **if** (<expression>) <statement> **else** <statement>

The second extension is the use of braces in an RHS to indicate that the enclosed part can be repeated indefinitely or left out altogether

BNF and EBNF Versions of an Expression Grammar

BNF:
<expr> → <expr> + <term>
| <expr> - <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>
<factor> → <exp> ** <factor>
<exp>
<exp> → (<expr>)
| id

EBNF:
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
| id

4 .Attribute Grammars:

An **attribute grammar** is a device used to describe more of the structure of a programming language than can be described with a context-free grammar. An attribute grammar is an extension to a context-free grammar. The extension allows certain language rules to be conveniently described, such as type compatibility.

- It is a bottom up apporch.

4.1 Static Semantics:

There are some characteristics of the structure of programming languages that are difficult to describe with BNF, and some that are impossible.

an example of a syntax rule that is difficult to specify with BNF, consider type compatibility rules. In Java, for example, a floating-point value cannot be assigned to an integer type variable, although the opposite is legal

If all of the typing rules of Java were specified in BNF, the grammar would become too large to be useful, because the size of the grammar determines the size of the syntax analyzer.

The **static semantics** of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics).

Static semantics is so named because the analysis required to check these specifications can be done
at compile time.

Attribute grammars are a formal approach both to describing and checking the correctness of the static semantics rules of a program.

4.2 Attribute Grammars Defined:

An attribute grammar is a grammar with the following additional features:
There are two type :

- **Synthesized attributes.**
- **Inherited attributes**.

Synthesized attributes:

Synthesized attributes are used to pass semantic information up a parse tree,

- ✓ Associated with each grammar symbol X is a set of attributes A(X).
- ✓ consists of one sets S(X) it is a represationent.
- ✓ the attributes of the symbols in the grammar rule.For a rule X_0 □ X_1 …… X*n*, the synthesized attributes of X_0 are computed with semantic functions of the form S(X0) = f(A(X1), …. , A(X*n*)).
- ✓ So the value of a synthesized attribute on a parse tree node depends only on the values of the attributes on that node's children nodes.

Inherited attributes:

A inherited attributes pass semantic information down and across a tree,

- ✓ Associated with each grammar symbol X is a set of attributes A(X).
- ✓ consists of one sets S(X) it is a represationent.
- ✓ Inherited attributes of symbols X*j*, 1 <= *j* <= *n* (in the rule above), are computed with a semantic function of the form I(X*j*) = f(A(X0), …. , A(X*n*)).

✓ So the value of an inherited attribute on a parse tree node depends on the attribute values of that node's parent node and those of its sibling nodes.

4.3Intrinsic Attributes:

Intrinsic attributes are synthesized attributes of leaf nodes whose values are determined outside the parse tree.

For example, the type of an instance of a variable in a program could come from the symbol table, which is used to store variable names and their types The contents of the symbol table are set based on earlier declaration statements.

4.4 Examples of Attribute Grammars:

The attribute grammars can be used to describe static semantics, consider the following fragment of an attribute grammar that describes the rule that the name on the end of an Ada procedure must match the procedure's name.

The string attribute of <proc_name>, denoted by <proc_name>.string, is the actual string of characters that were found immediately following the reserved word procedure by the compiler.

Neither the subscripts nor the brackets are part of the described language.

Syntax rule: <proc_def> → procedure <proc_name>[1]
<proc_body> end <proc_name>[2];

Predicate: <proc_name>[1] string == <proc_name>[2] .string

- **actual_type**—A synthesized attribute associated with the nonterminals <var> and <expr>. It is used to store the actual type, int or real, of a variable or expression. In the case of a variable, the actual type is intrinsic. In the case of an expression, it is determined from the actual types of the child node or children nodes of the <expr> nonterminal.

- **expected_type**—An inherited attribute associated with the nonterminal <expr>. It is used to store the type, either int or real, that is expected for the expression, as determined by the type of the variable on the left side of the assignment statement.

The complete attribute grammar follows

For example of Simple Assignment Statements :

1. Syntax rule: <assign> → <var> = <expr>

   Semantic rule: <expr>.expected_type ← <var>.actual_type

2. Syntax rule: <expr> → <var>[2] + <var>[3]

   Semantic rule: <expr>.actual_type ←

   if (<var>[2].actual_type = int) and

   (<var>[3].actual_type = int)

   then int

   else real

   end if

   Predicate: <expr>.actual_type == <expr>.expected_type

3. Syntax rule: <expr> → <var>

   Semantic rule: <expr>.actual_type ← <var>.actual_type

   Predicate: <expr>.actual_type == <expr>.expected_type

4. Syntax rule: <var> → A | B | C

   Semantic rule: <var>.actual_type ← look-up(<var>.string)

Lexical Analysis:

A lexical analyzer is essentially a pattern matcher. A pattern matcher attempts to find a substring of a given string of characters that matches a given character pattern. Pattern matching is a traditional part of computing

such as the ed line editor, which was introduced in an early version of UNIX. Since then, pattern matching has found its way into some programming languages—for example, Perl and JavaScript. It is

also available through the standard class libraries of Java, C++, and C#.

- A lexical analyzer serves as the front end of a syntax analyzer. Technically, lexical analysis is a part of syntax analysis.
- A lexical analyzer performs syntax analysis at the lowest level of program structure. An input program appears to a compiler as a single string of characters.

- The lexical analyzer collects characters into logical groupings and assigns internal codes to the groupings according to their structure.

- these logical groupings are named **lexemes**,& the internal codes for categories of these groupings are named **tokens**.

- Lexemes are recognized by matching the input character string against character string patterns

- Although tokens are usually represented as integer values, for the sake of readability of lexical and syntax analyzers, they are often referenced through named constants.
  example of an assignment statement:

  result = oldsum – value / 100;

  The tokens and lexemes of this statement:

Token	*Lexeme*
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

- Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens.

- In the early days of compilers, lexical analyzers often processed an entire source program file and produced a file of tokens and lexemes.

- lexical analyzers are subprograms that locate the next lexeme in the input, determine its associated token code, and return them to the caller, which is the syntax analyzer.

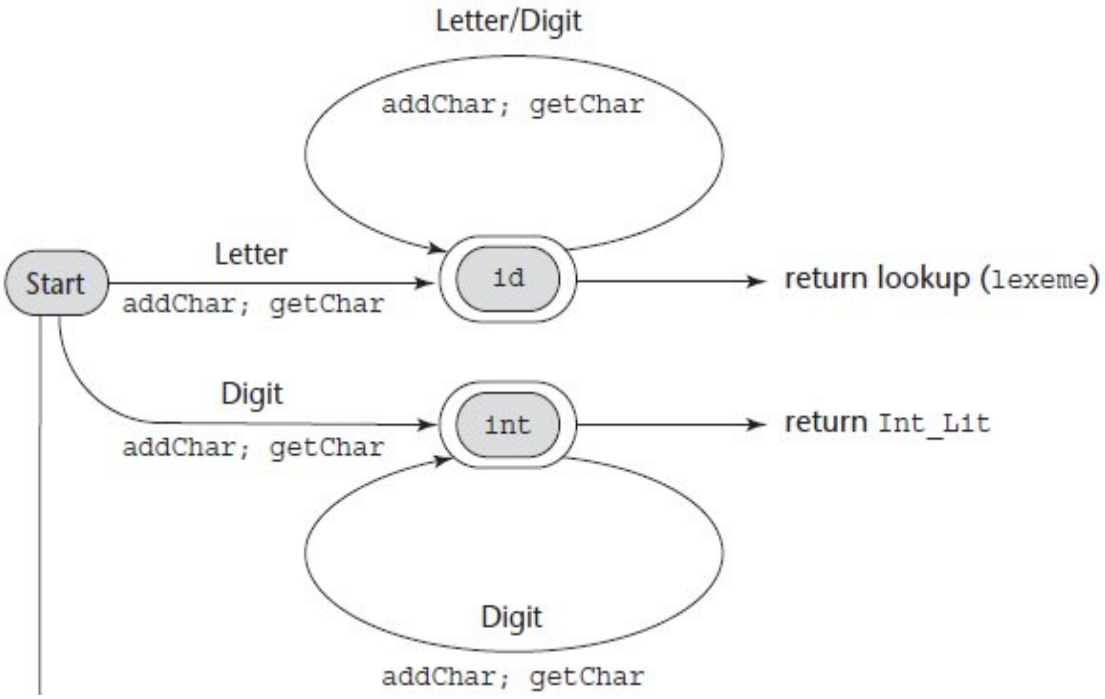
- each call to the lexical analyzer returns a single lexeme and its token. The only view of the input program seen by the syntax analyzer is the output of the lexical analyzer, one token at a time.

Three approaches to building a lexical analyzer:

1. Write a formal description of the token patterns of the language usinga descriptive language related to regular expressions. These descriptions are used as input to a software tool that automatically generates a nlexical analyzer. There are many such tools available for this. The oldest of these, named lex, is commonly included as part of UNIX systems.
2. Design a state transition diagram that describes the token patterns ofthe language and write a program that implements the diagram.
3. Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram

```
/* Parse identifiers */
case LETTER:
addChar();
getChar();
while (charClass == LETTER || charClass == DIGIT) {
addChar();
getChar();
}
nextToken = IDENT;
break;
/* Parse integer literals */
case DIGIT:
addChar();
getChar();
while (charClass == DIGIT) {
addChar();
getChar();
}
nextToken = INT_LIT;
break;
```

The state diagram:



UNIT-1

Parsing:

The part of the process of analyzing syntax that is referred to as *syntax analysis* is often called **parsing.**

Two main categories of parsing algorithms, top-down and bottom-up, as well as the complexity of the parsing process.

- Parsers for programming languages construct parse trees for given programs. In some cases, the parse tree is only implicitly constructed, meaning that perhaps only a traversal of the tree is generated.
- all cases, the information required to build the parse tree is created during the parse.
- Both parse trees and derivations include all of the syntactic information needed by a language processor.
- the syntax analyzer must check the input program to determine whether it is syntactically correct.
- When an error is found, the analyzer must produce a diagnostic message and recover. In this case, recovery means it must get back to a normal state and continue its analysis of the input program.
- This step is required so that the compiler finds as many errors as possible during a single analysis of the input program. If it is not done well, error recovery may create more errors, or at least more error messages.
- The second goal of syntax analysis is to produce a complete parse tree, or at least trace the structure of the complete parse tree, for syntactically correct input. The parse tree (or its trace) is used as the basis for translation.

For formal languages, they are as follows:

1. Terminal symbols—lowercase letters at the beginning of the alphabet (a, b, . . .)

2. Nonterminal symbols—uppercase letters at the beginning of the alphabet (A, B, . . .)

4. Terminals or nonterminals—uppercase letters at the end of the alphabet (W, X, Y, Z)

5. Strings of terminals—lowercase letters at the end of the alphabet (w, x,y, z)

6. Mixed strings (terminals and/or nonterminals)—lowercase Greek letters (_, _, _, _)

Top-Down Parsers:

A top-down parser traces or builds a parse tree in preorder. A preorder traversal of a parse tree begins with the root.

Each node is visited before its branches are followed. Branches from a particular node are followed in left-to-right order. This corresponds to a leftmost derivation.

A top-down parser can be described as follows:

- Given a sentential form that is part of a leftmost derivation, the parser's task is to find the next sentential form in that leftmost derivation.
- The general form of a left sentential form is xA_, whereby our notational conventions x is a string of terminal symbols
- A is a nonterminal, and _ is a mixed string. Because x contains only terminals,
- A is the leftmost nonterminal in the sentential form, so it is the one that must be expanded to get the next sentential form in a leftmost derivation.
- Determining the next sentential form is a matter of choosing the correct grammar rule that has A as its LHS.

For example, if the current sentential form is

$$xA\alpha$$

The A-rules are $A \rightarrow bB$, $A \rightarrow cBb$, and $A \rightarrow a$, a top-down parser must choose among these three rules to get the next sentential form, which could be $xbB\alpha$, $xcBb\alpha$, or $xa\alpha$. This is the parsing decision problem for top-down parsers.

Bottom-Up Parsers:

A bottom up parser constructs a parse tree by beginning at the leaves and progressing toward the root.

This parse order corresponds to the reverse of a rightmost derivation. That is, the sentential forms of the derivation are produced in order of last to first.

In terms of the derivation, a bottom-up parser can be described as follows:

- Given a right sentential form α, the parser must determine what substring of _ is the RHS of the rule in the grammar that must be reduced to its LHS to produce the previous sentential form in the rightmost derivation.

- the first step for a bottom-up parser is to determine which substring of the initial given sentence is the RHS to be reduced to its corresponding LHS

- To get the second last sentential form in the derivation. The process of finding the correct RHS to reduce is complicated by the fact that a given right sentential form may include more than one RHS

- From the grammar of the language being parsed. The correct RHS is called the **handle**.

Consider the following grammar and derivation:

S → aAc
A → aA _ b
S => aAc => aaAc => aabc

- ✓ A bottom-up parser of this sentence, aabc, starts with the sentence and must find the handle in it.
- ✓ In this example, this is an easy task, for the string containsonly one RHS, b.
- ✓ When the parser replaces b with its LHS, A, it gets the second to last sentential form in the derivation, aaAc.
- ✓ In the general case, as stated previously, finding the handle is much more difficult, because a sentential form may include several different RHSs
- ✓ A bottom-up parser finds the handle of a given right sentential form by examining the symbols on one or both sides of a possible handle.

Complexity of Parsing:

Parsing algorithms that work for any unambiguous grammar are complicated and inefficient. In fact, the complexity of such algorithms is O(n3).

which means the amount of time they take is on the order of the cube of the length of the string to be parsed.

This relatively large amount of time is required because these algorithms frequently must back up and reparse part of the sentence being analyzed.

Recursive-Descent Parsing:

A recursive-descent parser is so named because it consists of a collection of subprograms, many of which are recursive, and it produces a parse tree in top-down order.

recursion is a reflection of the nature of programming languages, which include several different kinds of nested structures

For example, statements are often nested in other statements. Also, parentheses in expressions must be properly nested. The syntax of these structures is naturally described with recursive grammar rules.

EBNF is ideally suited for recursive-descent parsers.

which specify that what they enclose can appear zero or more times, and brackets, which specify that what they enclose can appear once or not at all.

Consider the following examples:

<if_statement> → **if** <logic_expr> <statement> [**else** <statement>]
<ident_list> → ident {, ident}

In the first rule, the **else** clause of an **if** statement is optional. In the second, an <ident_list> is an identifier, followed by zero or more repetitions of a commaand an identifier.

A recursive-descent parser has a subprogram for each nonterminal in its associated grammar the subprogram associated with a particular nonterminal is as follows When given an input string, it tracesout the parse tree that can be rooted at that nonterminal and whose leavesmatch the input string. In effect, a recursive-descent parsing subprogram is a parser for the language (set of strings) that is generated by its associated nonterminal.

EBNF description of simple arithmetic expressions:

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | (<expr>)

That an EBNF grammar for arithmetic expressions, such as this one, does not force any associativity rule. Therefore, when using such a grammar as the basis for a compiler, one must take care to ensure that the code generation process, which is normally driven by syntax analysis, produces code that adheres to the associativity rules of the language.

Bottom-Up Parsing:

This section introduces the general process of bottom-up parsing and includes a description of the LR parsing algorithm.

UNIT-1