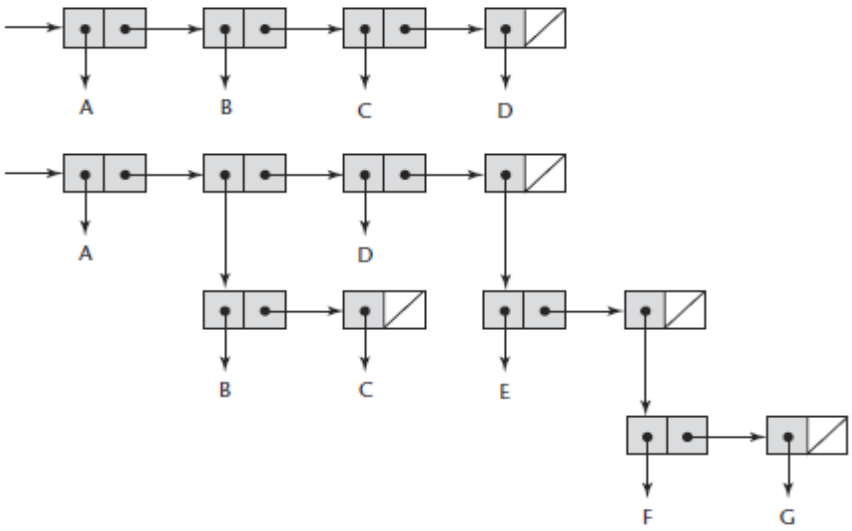
UNIT-I

Evolution of Programming Language

LISP:

- The name *LISP* derives from "LISt Processing". It was invented by John McCarthy (IBM Information Research Department) in 1958.His goal was to investigatesymbolic computations and to develop a set of requirements for doing suchcomputations.
- Linked lists are one of Lisp language's major data structures, and Lispsource code is itself made up of lists.
- The first version of LISP is sometimes called “pure LISP” because it is a purelyfunctional language.
- Pure LISP has only two kinds of data structures: atoms and lists. Atoms areeither symbols, which have the form of identifiers, or numeric literals.The concept of storing symbolic information in linked lists is natural and was used inIPL-II. Such structures allow insertions and deletions at any point, operationsthat were then thought to be a necessary part of list processing.
- Lists are specified by delimiting their elements with parentheses. Simplelists, in which elements are restricted to atoms, have the form(A B C D).
- Nested list structures are also specified by parentheses. For example, the list
  (A (B C) D (E (F G)))
  is composed of four elements. The first is the atom A; the second is the sublist(B C); the third is the atom D; the fourth is the sublist (E (F G)), which hasas its second element the sublist(F G).
- Internally, lists are stored as single-linked list structures, in which eachnode has two pointers and represents a list element.



Internal representation of two LISP lists

- LISP expressions are composed of *forms*. The most common LISP form is *function application*. LISP represents a function call *f(x)* as (f x). For example, *cos(0)* is written as (cos 0).
- LISP expressions are case-insensitive.
- (+ x1 x2……xn) – The sum of x1, x2, ….,xn.
- (* x1 x2……xn) – The product of x1, x2, ….,xn.
- (- x y) – subtract y from x.
- (/ x y) – divide x by y.

FORTRAN:

- The FORTRAN derives from "FORmula TRANslation". It is the oldest language still in use created by *John Backus in 1951*
- It is developed to perform high -.level scientific, mathematical, statistical computations.
- It is still used in aerospace, automotive industries, government and research institutions.

The environment in which the FORTRAN was developed was as follows:

- Computers had small memories and were slow and relatively unreliable.
- The primary use of computers was for scientific computations.
- Cost of computers are high while compare to cost of programmers.

FORTRAN 0 – 1954, FORTRAN 1 – 1955, FORTRAN 2 – 1958,

FORTRAN IV – 1977, 90, 95, 2003, 2008.

Sample Program:

(1.) Hello World

Ouput for Hellow World

```
WRITE(6,*)'Hello world'
      STOP
      END
```

(2.)!My first program

```
program first
print *,'This is  my first program'
end program  first.
```

ALGOL 60:

- ALGOL (ALGOrithmic Language) is one of several high level languages designed specifically for programming scientific computations. It started out in the late 1950's, first formalized in a report titled ALGOL 58, and then progressed through reports ALGOL 60, and ALGOL 68.
- ALGOL was the first second-generation programming language and its characteristics are typical of the entire generation.
- First consider the data structures, which are very close to first generation structures.
- In ALGOL 60 the block structure was introduced: the ability to create blocks of statements for the scope of variables and the extent of influence of control statements.
- Along with that, two different means of passing parameters to subprograms; call by value and call by name.
- Structured control statements: if - then - else and the uses of a general condition for iteration control were also features, as was the concept of recursion: the ability of a procedure to call itself.

- **Dynamic Arrays** - one for which the subscript range is specified by variables so that the size of the array is set at the time storage is allocated.
- **Reserved Words** - the symbols used for keywords are not allowed to be used as identifiers by the programmer.
- **User defined data types** - allow the user to design data abstractions that fit particular problems very closely.

Source Code: Hello World

```
// the main program (this is a comment)

    BEGIN

    FILE F (KIND=REMOTE);

    EBCDIC ARRAY E [0:11];

    REPLACE E BY "HELLO WORLD!";

    WHILE TRUE DO

    BEGIN

    WRITE (F, *, E);

    END;

    END.
```

COBOL:

- The name means **Common Business Oriented Language** . COBOL is referred to as a**legacy** language, which means it is in a format that is no longer used or supported by new systems.
- It was one of the first high-level programming languages created. COBOL is run on the mainframe as well as on the PC.
- we can write COBOL programs in text editors like Notepad or Notepad++. Once it is written, the program must be compiled to check for errors and converted into a language that the computer can read.

Divisions of COBOL:

The first thing to understand is that COBOL is divided into four divisions. The divisions are created in the program in this order:

1. Identification Division
2. Environment Division
3. Data Division
4. Procedure Division

Source Code : Hello World

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.HELLOWORLD.
000300
000400*
000500 ENVIRONMENT DIVISION.
000600 CONFIGURATION SECTION.
000700 SOURCE-COMPUTER. RM-COBOL.
000800 OBJECT-COMPUTER. RM-COBOL.
000900
001000 DATA DIVISION.
001100 FILE SECTION.
001200
100000 PROCEDURE DIVISION.
```

```
100100
100200 MAIN-LOGIC SECTION.
100300 BEGIN.
100400     DISPLAY " " LINE 1 POSITION 1 ERASE EOS.
100500     DISPLAY "Hello world!" LINE 15 POSITION 10.
100600     STOP RUN.
100700 MAIN-LOGIC-EXIT.
100800     EXIT.
```

BASIC:

- **BASIC** (an acronym for **Beginner's All-purpose Symbolic Instruction Code**) is a family of general-purpose, high-level programming languages whose design philosophy emphasizes ease of use.
- In 1964, John G. Kemeny and Thomas E. Kurtz designed the original BASIC language at Dartmouth College in New Hampshire. They wanted to enable students in fields other than science and mathematics to use computers.
- It is **completely free** and it is suitable for creating all kinds of applications for business, industry, education and entertainment.

The goals of the systemwere as follows:

- It must be easy for nonscience students to learn and use.
- It must be "pleasant and friendly."
- It must provide fast turnaround for homework.
- It must allow free and private access.
- It must consider user time more important than computer time.

Source Code:

```
10 PRINT "Hello World!"
20 GOTO 10
//This program prints the phrase "Hello World" infinitely.
```

Sample Run

```
Hello World!
Hello World!
Hello World!  etc., etc....
```

Describing Syntax

Syntax - the form or structure of the expressions, statements, and program units.
Semantics - the meaning of the expressions, statements, and program units.
Ex:

while (<Boolean_expr>)<statement>

- The semantics of this statement form is that when the current value of the Boolean expression is true, the embedded statement is executed.
- The form of a statement should strongly suggest what the statement is meant to accomplish.

The General Problem of Describing Syntax:

- A language, whether natural (such as English) or artificial (such as Java), is a setof strings of characters from some alphabet.The strings of a language are calledsentences or statements.
- A **sentence** "statement" is a string of characters over some alphabet. The syntax rules of a language specify which strings of characters from the language's alphabet are in the language.
- A **language** is a set of sentences.
- A **lexeme** is the lowest level syntactic unit of a language. It includes identifiers, literals, operators, and special word. (e.g. *, sum, begin) A program is strings of lexemes. The lexemes of a programming language include its numeric literals, operators, and special words, among others.
- Lexemes are partitioned into groups—for example, the names of variables,methods, classes, and so forth in a programming language form a group called identifiers.
- A **token** is a category of lexemes (e.g., identifier.) An identifier is a token that have lexemes, or instances, such as sum and total.
- Ex:
  index = 2 * count + 17;

Lexemes	***Tokens***
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon.

Formal Methods of Describing Syntax:

Backus-Naur Form and Context-Free Grammars

- It is a syntax description formalism that became the mostly wide used method for P/L syntax.

Context-free Grammars:

- Developed by Noam Chomsky in the mid-1950s who described four classes of generative devices or grammars that define four classes of languages.
- Context-free and regular grammars are useful for describing the syntax of P/Ls.
- Tokens of P/Ls can be described by regular grammars.
- Whole P/Ls can be described by context-free grammars.

Backus-Naur Form (1959):

- Invented by John Backus to describe ALGOL 58 syntax.
- **BNF** is equivalent to context-free grammars used for describing syntax.

Fundamentals

- A metalanguage is a language used to describe another language "Ex: BNF."
- In **BNF**, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called nonterminal symbols)

  <while_stmt>→ while (<logic_expr>) <stmt>

- This is a rule; it describes the structure of a while statement
- A rule has a left-hand side (LHS) "The abstraction being defined" and a right-hand side (RHS) "consists of some mixture of tokens, lexemes and references to other abstractions", and consists of terminal and nonterminal symbols.
- A grammar is a finite nonempty set of rules and the abstractions are called **nonterminal symbols**, or simply **nonterminals**.
- The lexemes and tokens of the rules are called **terminal** symbols or **terminals**.
- An abstraction (or nonterminal symbol) can have more than one RHS

  <stmt>→<single_stmt>

  | begin <stmt_list> end

- Multiple definitions can be written as a single rule, with the different definitions separated by the symbol |, meaning logical **OR**.

Describing Lists:

- Syntactic lists are described using recursion.

<ident_list>→ident

| ident, <ident_list>

- A rule is **recursive** if its LHS appears in its RHS.

Grammars and derivations:

- 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**.
- A **derivation** is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
- An example grammar:

<program>→<stmts>

<stmts>→<stmt> | <stmt> ; <stmts>

<stmt>→<var> = <expr>

<var>→ a | b | c | d

<expr>→<term> + <term> | <term> - <term>

<term>→<var> | const

- An example derivation:

<program> =><stmts> =><stmt>

=><var> = <expr> => a = <expr>

=> a = <term> + <term>

=> a = <var> + <term>

=> a = b + <term>

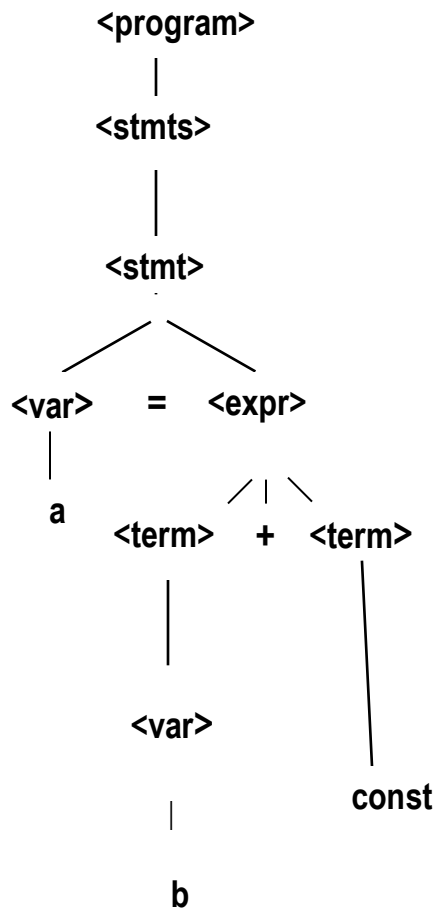
=> a = b + const

- Every string of symbols in the derivation, including <program>, is a **sentential form**.
- A sentence is a sentential form that has only terminal symbols.

- A **leftmost derivation** is one in which the leftmost nonterminal in each sentential form is the one that is expanded. The derivation continues until the sentential form contains no nonterminals.
- A derivation may be neither leftmost nor rightmost.

Parse Trees:

- Hierarchical structures of the language are called **parse trees**..
- A parse tree for the simple statement A = B + const



Ambiguity:

- A grammar is ambiguous if it generates a sentential form that has two or more distinct parse trees.
- Ex: Two distinct parse trees for the same sentence, const – const / const

  <expr>→<expr><op><expr> | const

  <op>→ / | -

Ex: Two distinct parse trees for the same sentence, A = B + C * A

<assign>→ <id> = <expr>

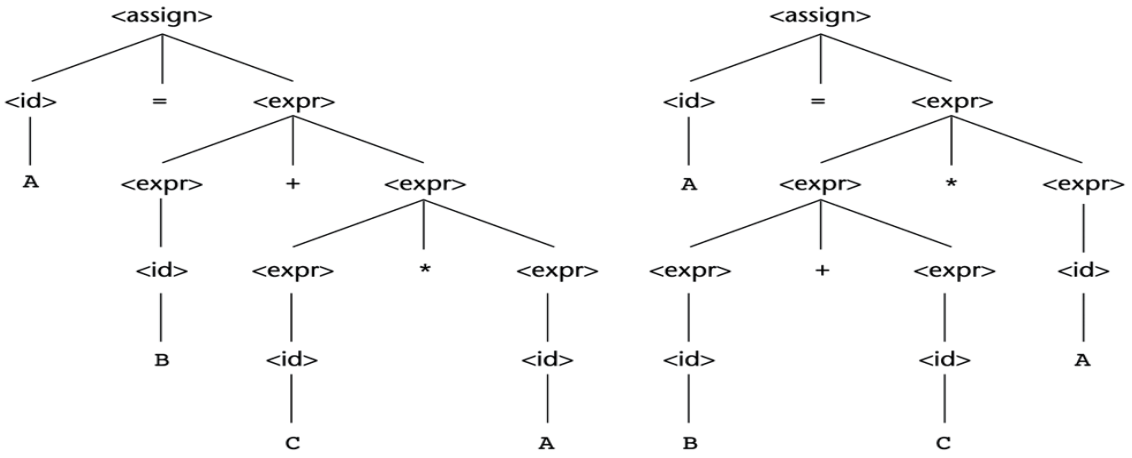
<id> → A | B | C

<expr>→ <expr> + <expr>

| <expr> * <expr>

| (<expr>)

| <id>



Operator Precedence:

- The fact that an operator in an arithmetic expression is generated lower in the parse tree can be used to indicate that it has higher precedence over an operator produced higher up in the tree.
- In the left parsed tree above, one can conclude that the * operator has precedence over the + operator. How about the tree on the right hand side?
- An unambiguous Grammar for Expressions

<assign>→ <id> = <expr>

<id> → A | B | C

<expr>→ <expr> + <term>

| <term>

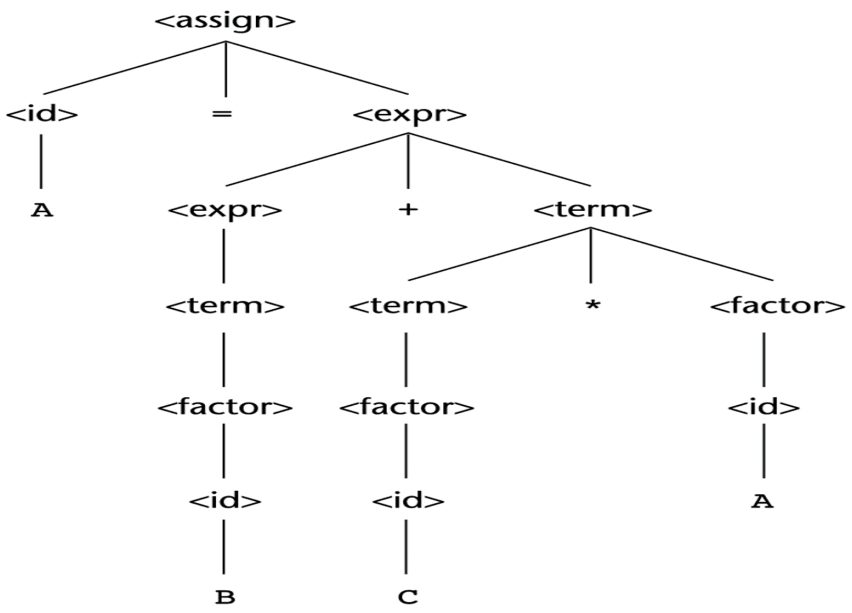
<term>→ <term> * <factor>

| <factor>

<factor>→ (<expr>)

| <id>

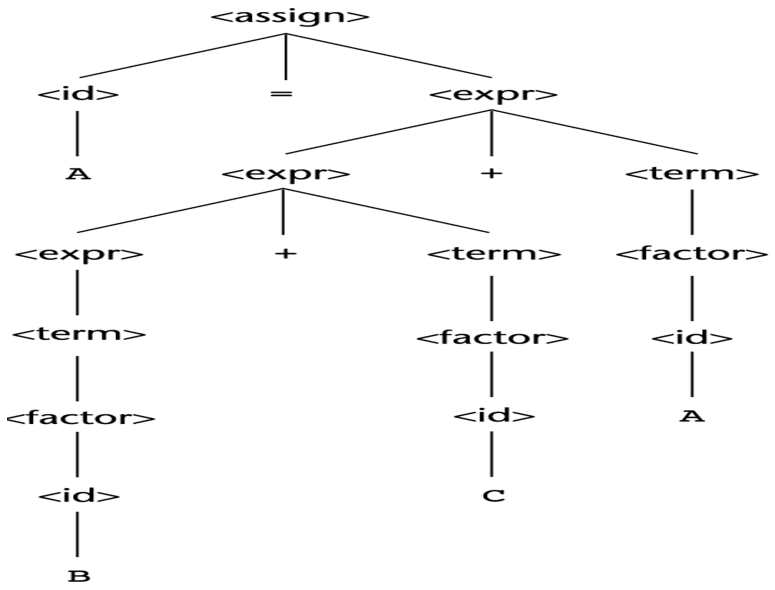
A = B + C * A



Associativity of Operators:

- Do parse trees for expressions with two or more adjacent occurrences of operators with equal precedence have those occurrences in proper hierarchical order?
- An example of an assignment using the previous grammar is:

A = B + C + A



- Figure above shows the left + operator lower than the right + operator. This is the correct order if + operator meant to be left associative, which is typical.

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.
- Attribute grammars are a formal approach both to describing and checking the correctness of the static semantics rules of a program.
- 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).
- Dynamic semantics, which is the meaning of expressions, statements, and program units.
- Attribute grammars are context-free grammars to which have been added attributes,attribute computation functions, and predicate functions.
- Attribute computation functions, sometimes called semanticfunctions, are associated with grammar rules. They are used to specify howattribute values are computed.
- Predicate functions, which state the staticsemantic rules of the language, are associated with grammar rules.

The syntax portion of our example attribute grammar is

<assign>-><var> = <expr>
<expr>-><var> + <var>
| <var>
<var>-> A | B | C

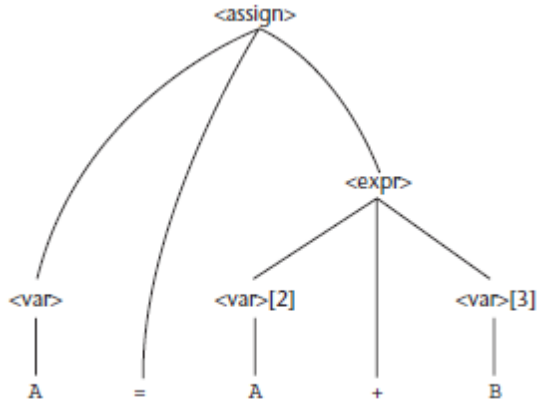
The attributes for the nonterminals in the example attribute grammar aredescribed in the following paragraphs:

- *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 orexpression. 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 nodeor 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 forthe expression, as determined by the type of the variable on the left side of the assignment statement.

An Attribute Grammar for 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)
thenint
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).

Parse Tree: A= A + B



Example

$E \rightarrow E + T$	$E_1 \rightarrow E_2 + \mathtt{T}$	$\triangleright E_1.val := \text{sum}(E_2.val, T.val)$
$E \rightarrow E - T$	$E_1 \rightarrow E_2 - \mathtt{T}$	$\triangleright E_1.val := \text{difference}(E_2.val, T.val)$
$E \rightarrow T$	$E \rightarrow T$	$\triangleright E.val := T.val$
$T \rightarrow T * F$	$T_1 \rightarrow T_2 * F$	$\triangleright T_1.val := \text{product}(T_2.val, F.val)$
$T \rightarrow T / F$	$T_1 \rightarrow T_2 / F$	$\triangleright T_1.val := \text{quotient}(T_2.val, F.val)$
$T \rightarrow F$	$T \rightarrow F$	$\triangleright T.val := F.val$
$F \rightarrow -F$	$F_1 \rightarrow -F_2$	$\triangleright F_1.val := \text{negate}(F_2.val)$
$F \rightarrow (E)$	$F \rightarrow (E)$	$\triangleright F.val := E.val$
$F \rightarrow const$	$F \rightarrow const$	$\triangleright F.val := const.val$

Synthesized Attributes (Bottom-Up):

- The language

$$\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 1\} = \{\mathtt{abc}, \mathtt{aabbcc}, \mathtt{aaabbbccc}, \ldots\}$$

is not context-free but can be recognized by an attribute grammar.

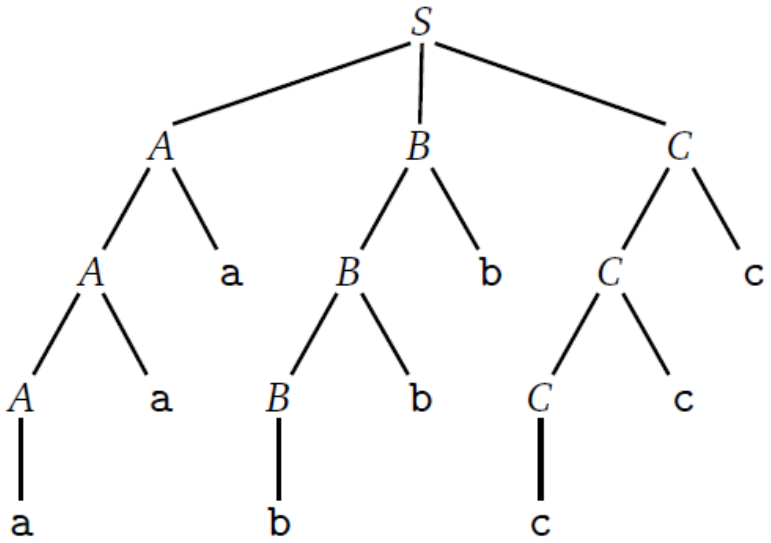
- Attributes of LHS are computed from attributes ofRHS.

$S \rightarrow ABC$	$\triangleright$ Condition: $A.count = B.count = C.count$
$A \rightarrow \mathtt{a}$	$\triangleright A.count := 1$
$A_1 \rightarrow A_2\ \mathtt{a}$	$\triangleright A_1.count := A_2.count + 1$
$B \rightarrow \mathtt{b}$	$\triangleright B.count := 1$
$B_1 \rightarrow B_2\ \mathtt{b}$	$\triangleright B_1.count := B_2.count + 1$
$C \rightarrow \mathtt{c}$	$\triangleright C.count := 1$
$C_1 \rightarrow C_2\ \mathtt{c}$	$\triangleright C_1.count := C_2.count + 1$

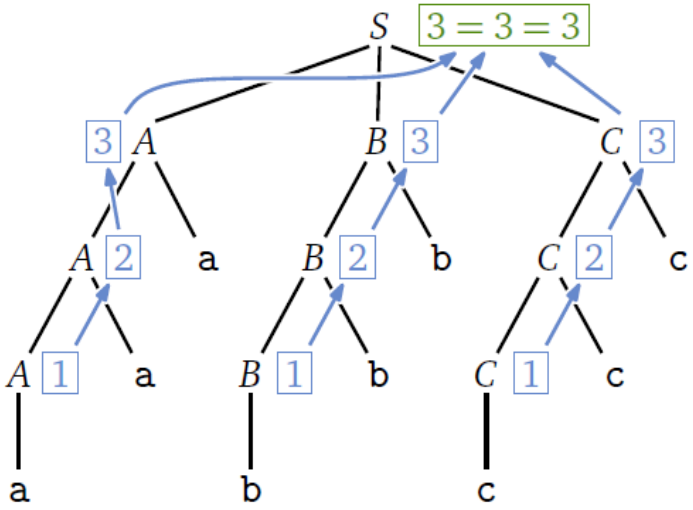
Input: aaabbbccc

Parse Tree:

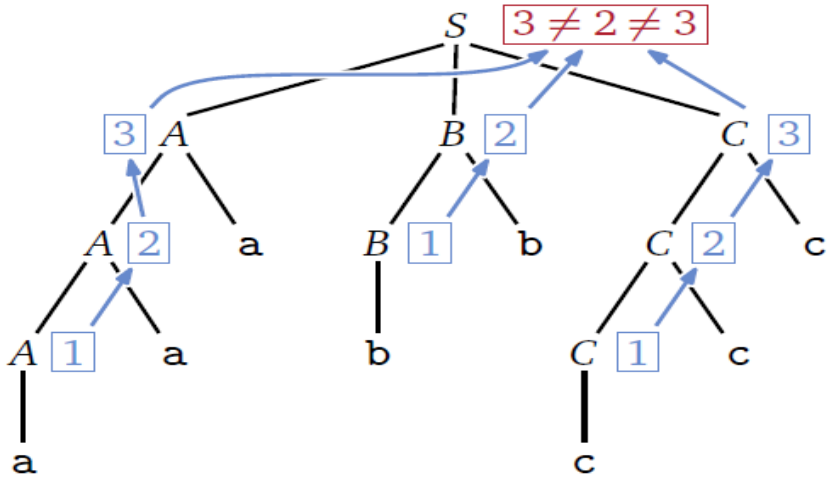
Normal Parse Tree



Input: aaabbbcccSynthesized Parse Tree



Input: aaabbccc

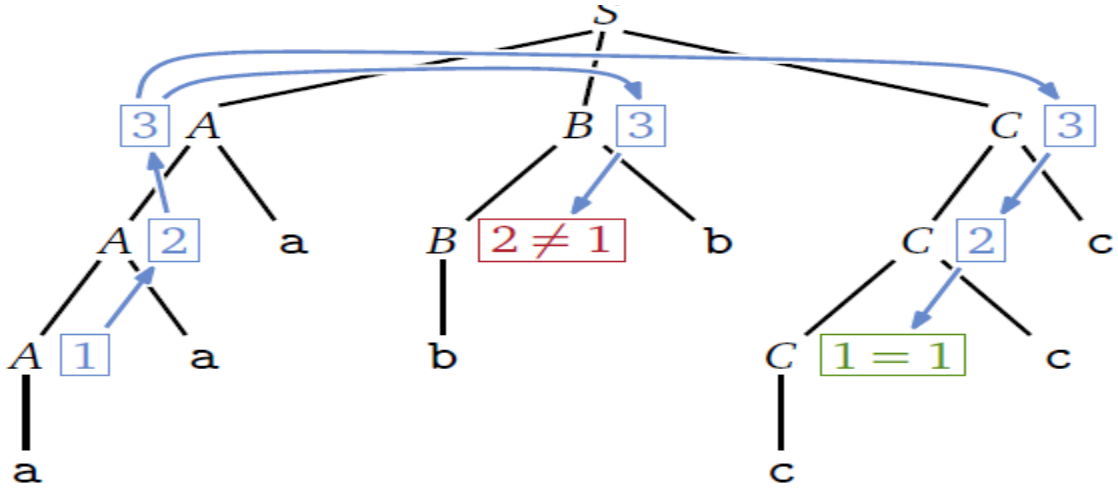


Inherited Attributes (Top-Down):

- Attributes flow from left to right (from LHS to RHS and from symbols of RHS to symbols of RHS further to the right).

$S \rightarrow ABC$	$\triangleright\ B.inhCount := A.count; C.inhCount := A.count$
$A \rightarrow \mathtt{a}$	$\triangleright\ A.count := 1$
$A_1 \rightarrow A_2\ \mathtt{a}$	$\triangleright\ A_1.count := A_2.count + 1$
$B \rightarrow \mathtt{b}$	$\triangleright$ Condition: $B.inhCount = 1$
$B_1 \rightarrow B_2\ \mathtt{b}$	$\triangleright\ B_2.inhCount := B_1.inhCount - 1$
$C \rightarrow \mathtt{c}$	$\triangleright$ Condition: $C.inhCount := 1$
$C_1 \rightarrow C_2\ \mathtt{c}$	$\triangleright\ C_2.inhCount := C_1.inhCount - 1$

Input: aaabbccc



Describing Semantics

Describing the Meanings of Programs: Dynamic Semantics

1.) Axiomatic Semantics:

- Axiomatic Semantics was defined in conjunction with the development of a method to prove the correctness of programs.
- Such correction proofs, when they can be constructed, show that a program performs the computation described by its specification.
- In a proof, each statement of a program is both preceded and followed by a logical expression that specified constraints on program variables.
- Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions.)
- The expressions are called **assertions**.

Assertions:

- Axiomatic semantics is based on mathematical logic. The logical expressions are called predicates, or **assertions**.
- An assertion before a statement (a **precondition**) states the relationships and constraints among variables that are true at that point in execution.
- An assertion following a statement is a **postcondition**.
- A **weakest precondition** is the least restrictive precondition that will guarantee the validity of the associated postcondition.

Pre-post form: {P} statement {Q}

An example: a = b + 1 {a > 1}

One possible precondition: {b > 10}
Weakest precondition: {b > 0}

- If the **weakest precondition** can be computed from the given postcondition for each statement of a language, then correctness proofs can be constructed from programs in that language.

2.) Operational Semantics:

- **operational semantics** is to describe the meaning of a statement or program by specifying the effects of running it on a machine.

- Most programmers have, on at least one occasion, written a small test program to determine the meaning of some programming language construct, often while learning the language.
- There are different levels of uses of operational semantics.
  - At the highest level, the interest is in the final result of the execution of a complete program. This is sometimes called natural operational semantics
  - At the lowest level,operational semantics can be used to determine the precise meaning of a program through an examination of the complete sequence of state changes that occur when the program is executed. This use is sometimes called structural operational semantics.

C Statement	*Meaning*

`for (expr1; expr2; expr3) {`
`  ...`
`}`
	`expr1;`
	`loop: if expr2 == 0 goto out`
	`...`
	`expr3;`
	`goto loop`
	`out: ...`

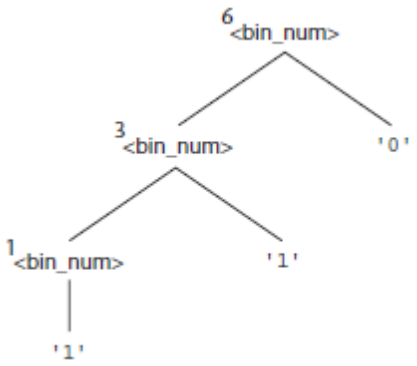
3.) Denotational Semantics:

- IT is the most rigorous and most widely known formal method for describing the meaning of programs.
- It is solidly based on recursive function theory.
- The process of constructing a denotational semantics specification for aprogramming language requires one to define for each language entity both amathematical object and a function that maps instances of that language entityonto instances of the mathematical object.
- The method is named denotational because the mathematical objects denote the meaning of their corresponding syntactic entities.
- In denotational semantics, the domain is called the syntactic domain, because it is syntactic structures that are mapped. The range is called the **semantic domain**.

Eg: The syntax of such binary numbers can be described by the following grammar rules:

<bin_num> → '0'
| '1'
| <bin_num> '0'
| <bin_num> '1'

A parse tree for the example binary number, 110



- The semantic function, named Mbin, maps the syntactic objects, as described in the previous grammar rules, to the objects in N, the set of nonnegative decimal numbers. The function Mbin is defined as follows:

$$M_{bin}('0') = 0$$
$$M_{bin}('1') = 1$$
$$M_{bin}(\langle bin\_num\rangle\ '0') = 2 * M_{bin}(\langle bin\_num\rangle)$$
$$M_{bin}(\langle bin\_num\rangle\ '1') = 2 * M_{bin}(\langle bin\_num\rangle) + 1$$

- A similar example for describing the meaning of syntactic decimal literals. In this case, the syntactic domain is the set of character string representations of decimal numbers. The semantic domain is once again the set N.

<dec_num> → '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7''8'|'9'
|<dec_num> ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')

The denotational mappings for these syntax rules are

$$M_{dec}('0') = 0, M_{dec}('1') = 1, M_{dec}('2') = 2, \ldots, M_{dec}('9') = 9$$
$$M_{dec}(\langle dec\_num\rangle\ '0') = 10 * M_{dec}(\langle dec\_num\rangle)$$
$$M_{dec}(\langle dec\_num\rangle\ '1') = 10 * M_{dec}(\langle dec\_num\rangle) + 1$$
$$\ldots$$
$$M_{dec}(\langle dec\_num\rangle\ '9') = 10 * M_{dec}(\langle dec\_num\rangle) + 9$$

Lexical Analysis

- The lexical analyzer deals with small-scale language constructs, such as names and numeric literals.
- The syntax analyzer deals with the large-scale constructs, such as expressions, statements, and program units.
- There are three reasons why lexical analysis is separated from syntax analysis:

- Simplicity—Techniques for lexical analysis are less complex than thoserequired for syntax analysis.
- Efficiency—Although it pays to optimize the lexical analyzer, because lexical analysis requires a significant portion of total compilation time, it is not fruitful to optimize the syntax analyzer.
- Portability—Because the lexical analyzer reads input program files and often includes buffering of that input, it is somewhat platform dependent.

- Lexical analysis is the extraction of individual words or lexemes from an input stream of symbols and passing corresponding tokens back to the parser.
- If we consider a statement in a programming language, we need to be able to recognize the small syntactic units (tokens) and pass this information to the parser. We need to also store the various attributes in the symbol or literal tables for later use, e.g., if we have an variable, the tokeniser would generate the token var and then associate the name of the variable with it in the symbol table - in this case, the variable name is the lexeme.
- Other roles of the lexical analyzer include the removal of whitespace and comments and handling compiler directives (i.e., as a preprocessor).
- The lexical analysis process starts with a definition of what it means to be a token in the language with regular expressions or grammars, then this is translated to an abstract computational model for recognizing tokens (a non-deterministic finite state automaton) which is then translated to an implementable model for recognising the defined tokens (a deterministic finite state automaton) to which optimisations can be made (a minimised DFA).

Parsing:

- The part of the process of analyzing syntax that is referred to as *syntax analysis* is often called **parsing**.
- The two main categories of parsing algorithms, top-down and bottom-up. Parsers are categorized according to the direction in which they build parse trees.
- The two broad classes of parsers are top-down, in which the tree is built from the root downward to the leaves, and bottom-up, in which the parse tree is built from the leaves upward to the root.
- Here we use a small set of notational conventions for grammar symbols and strings to make the discussion less cluttered. 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, . . .)
  3. Terminals or nonterminals—uppercase letters at the end of the alphabet (W, X, Y, Z)
  4. Strings of terminals—lowercase letters at the end of the alphabet (w, x,y, z)
  5. Mixed strings (terminals and/or nonterminals)—lowercase Greek letters $(\alpha, \beta, \delta, \gamma)$.

Example: As an example, let's trace through the two approaches on this simple grammar that recognizes strings consisting of any number of a's followed by at least one (and possibly more) b's:

S –> AB
A –> aA | ε
B –> b | bB

Top-Down Parsing:

- In top-down parsing,you start with the start symbol and apply the productions until you arrive at the desiredstring.
- Here is a top-down parse of aaab.

S	
AB	S –> AB
aAB	A –> aA
aaAB	A –> aA
aaaAB	A –> aA
aaaεB	A –> ε
aaab	B –> b

- A recursive-descent parser is a coded version of a syntax analyzer baseddirectly on the BNF description of the syntax of language.
- LL algorithms -The first L in LL specifies a left-to-right scan ofthe input; the second L specifies that a leftmost derivation is generated.

Bottom – Up Parsing:

- In bottom-up parsing, you start with the string and reduce it to the start symbolby applying the productions backwards.

aaab	
aaaεb	(insert ε)
aaaAb	A –> ε
aaAb	A –> aA
aAb	A –> aA
Ab	A –> aA
AB	B –> b
S	S –> AB

- The most common bottom-up parsing algorithms are in the LR family,where the L specifies a left-to-right scan of the input and the R specifies that arightmost derivation is generated.

Important Questions

1.) Explain about Evolution of Programming Languages?

2.) Define CFG? What does it mean for CFG to be ambiguous?

3.) Briefly explain about Attribute Grammar?

4.) Write about Describing Syntax and Semantics in detail?

5.) Explain about Lexical analysis in detail?

6.) Discuss in detail about parsing with an example?

7.) Using this grammar

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

Show the parse tree and Leftmost derivation for the following.

a.) A=(A+B)*C b.) A=B*(C*(A+B))

8.) What is CFG and prove the following grammar is ambiguous.

<S> →<A>
<A>→<A>+<A>|<id>
<id>-→a|b|c