Evolution of programming languages

- This focuses on environment in which each language was designed and on the contributions and motivations for their development.

- Zuse's plankalkul

- Between 1936 and 1945, German scientist Konrad Zuse, embarked on an effort to develop a language for expressing computions for his computer model called z4.
- In 1943, he named his language plankalkul meaning "program calculus". Zuse defined plankalkul and wrote algorithms in the language to solve a wide variety of problems.

⇒ plankalkul was remarkbly complete with many advanced features.

- The simplest data type was a single bit. Integer and floating-point numeric types were built from the bit-type.
- It included arrays and records (structs in c language).
- No goto statement, but had constructs similar to "for" and has a selective statement.
- The major disadvantage for its implementation is its notation. Each statement consisted of two or three lines of code.

- Pseudocodes

- These are languages that were developed and used in late 1940's and early 1950's.
- The computers that are available during this period were far less usable. They were slow, unreliable, expensive and had extremly small memories.

- There were no high-level programming languages or even assembly languages, so programming was done in machine code.
- Machine code programming is both tedious and error prone.
  - The code is difficult to read.
  - More serious problem is absolute addressing, which makes program modification tedious and error prone.
- These are standard problems with all machine languages and were the primary motivations for inventing assemblers and assembly languages.
- Most programming languages did not have floating-point arithmetic operations and indexing to allow the convenient use of arrays.

Short code

- Short code was developed in 1949, which was one of the first successful stored-program electronic computers.
- Short code was not translated to machine code, rather it was implemented with a pure interpreter. At the time, this process was called automatic programming

Speedcoding.

- The speed coding system developed for IBM 701 in 1954 is an example of interpretive systems that extended machine languages to include floating-point operations.
  - conditional and unconditional branches and I/o conversions were also present.

The UNIVAC "Compiling" System

- Between 1951 and 1953, UNIVAC developed a series of compiling systems named A-0, A-1 and A-2 that expanded a pseudocode into machinecode in the same way as macros are expanded into assembly language.
- pseudocode is an informal high-level description of a program or algorithm intended for human reading rather than machine reading.
- These pseudocode source for the compilers made the source programs much shorter.

⇒ FORTRAN =

- In 1954, language FORTRAN was invented by IBM. It was the first widely used high level general purpose programming language to have a functional implementation as opposed to just a design on paper.
- It is still popular language for high performance computing, used in world's fastest supercomputers
- It is especially suited for numeric computations and scientific computing.
- IBT 70H had both indexing and floating point instruction in hardware.
- The compilers were operational by this time.

⇒ LISP =

- The first functional programming language for performing list processing.
- To process symbolic data (language) in linked lists.

- LISP was designed as a functional programming language. All computation is purely functional and accomplished by applying functions to arguments.

ALGOL - 1958 - Start of sophesticated languages.

- ALGOL 60 has much influence on subsequent programming languages and is therefore of central importance in any historical study of languages.

Goals of these language were:

- The syntax of language should be as close as possible to standard mathematical notation. Should be readable with little further explanation.
- The programs in this new language must be mechanically translated into machine language.

So - The concept of block structure was introduce.
- Pass by value, pass by name were used for parameter passing.
- procedures were allowed to be recursive.
- dynamic arrays were allowed.

COBOL - Common Bussiness oriented language.

- It was the first application deigned for bussiness applications.
- The poor performance of early compilers simply made the language too expensive to use. Eventually compilers became efficient and computers became much faster and cheaper, which allowed COBOL to succeed.

BASIC - Begining of timeSharing. [Beginner's all purpouse Symbolic Instruction Code].

- Very popular on microcomputers in late 1970's and early 1980's.
- It was easy for beginners to learn, especially those who were not science oriented.
- It was the first widely used language that was used through terminals connected to a remote computer.
- Disadvantage is the poor structure of programs written in it.

PL/I - 1963

- It combined best parts of ALGOL 60, COBOL 60, Fortran IV
  - It had concurrently executing subprograms
  - It detects and handle 23 different types of exceptions or run time errors.
  - pointers were included as data types.

APL and SNOBOL → 1960's

- They provided dynamic typing and dynamic storage allocation.

SIMULA 67

- concept of data abstraction started with it.

ALGOL 68

- orthogonality design.
- orthogonality means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of a language.

PASCAL - Simple by design.

- By mid 1970's all universities started using PASCAL for learning programming.
- Basically pascal was designed as a teaching language.

C

- C was originally designed for system's programming.
- It is well suited for a wide variety of application.
- It was developed from ALGOL 68, CPL, BCPL, B. in 1972.
- Related languages - Objective C, Delphi,

Prolog - programming based on Logic - 1965

- Logic programming is the use of a formal logic notation to communicate computational procedures to a computer.
- predicate calculus is the notation used in current logic programming languages.
- It is non-procedural

Smalltalk: object orient programming.

- First language which supported object oriented programming.

C++ - 1980

- Combing function and object oriented features.
- Another related language is objective-C, which has object oriented features in C language.
- Relative languages Objective C, Delphi, Go.

Java - 1990

- Language to provide reliability and portable software that could be used in embedded systems.

- In 1993, with world wide web becoming widely used, Java was found to be the tool for web programming.
- Java's portability was also considered good feature for cody-platform interactions.
- Java designers eliminated the excess and unsafe features of C++.

Scripting languages

- A script is a file containing a list of commads, which is to be interpreted.

Perl

- perl was originally a combination of two languages of "sh" and "awk".
  - sh was a collection of commands that were interpreted to some utility functions such as file management and simple file filtering.
  - awk is a report-generation language.
  - perl combines these two language features to form a strong scripting language.

JavaScript

- with the use of web, need for computation associated with HTML documents, quickly became critical. performing the computation at the server side was also critical.
- Java Script helped in these problems.
- Initially it was named Livescript. Later SunMicrosystems named it JavaScript.

- JavaScript is a for dynamically creating and modifying HTML document.

- PHP - 2000.

- initially name personal home page, later named Hypertext preprocessor.
  - It is an HTML-embedded server-side scripting language specifically designed for web applications.

- Python - 2004
  - It is a object-oriented scripting language.
  - CGI programming, cookies, networking and database access is available as part of python.
  - It is easy to programming.

- Ruby - 2005
  - Both PHP and python are partly object-oriented languages.
  - Ruby was designed as fully object-oriented scripting language.

C# - The .Net language - 2002 - C Sharp
- C# along with .NET development platform was announced by microsoft in 2000.
- C# is based on C++ and Java but includes some ideas from delphi and visual Basic.

Markup/programming hybrid languages
- Programming hybrid languages is a markup language in which some of the elements can specify programming actions such as control flow and computation.

XSLT

XSLT - extensible markup language (XML) is a meta markup language.

- XML is used to define markup languages.
- XML documents are transformed into HTML or into other forms of data files.
- The transformation of XML documents to HTML document is specified in another markup language, extensible Stylesheet Language transformations.

JSP - Java Server Pages

- The core part of the Java Server pages Standard Tag Library (JSTL) is another markup/programming hybrid language.
- JSP contain Servelets and Java Server pages (JSP)
- JSP is a collection of technologies designed to support dynamic web documents and provide other processing needs of web documents

Unit - 1

Formal methods for describing Syntax.

- Formal language generation mechanisms usually called grammers, that are commonly used to describe the syntax of programming languages.

- 1950's, Chomsky described four classes of generative devices or grammers that define four classes of languages. Two of these grammer classes, named context-free and regular, are useful for describing the syntax of programming languages. other two are context-sensitive and unrestricted grammars.

- The forms of the tokens of programming language can be described by regular grammers. The syntax (Lexemes) of whole programming languages with minor exceptions can be described by context-free grammers.

BNF (Backus-Naur Form)

- BNF is a notation for describing syntax.
- A metalanguage is a language that is used to describe another language. BNF is a metalanguage for programming languages.
  - BNF uses abstractions for syntactic structures.

    Ex: A simple java assignment statement is represented by the abstraction <assign>. The actual definition is

    <assign> → <var> = <expression>

  - In this the text on the left hand side (LHS) is abstraction being defined. The text to the right of arrow (RHS) is the definition of LHS 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, the abstractions <var> and <expression> must defined for the <assign> definition to be useful.
- The abstractions in a BNF description or grammer are often called non-terminals or simply non-terminals. The lexemes and tokens of the rules are called terminal symbols or terminals.
- A BNF description or grammer is a collection of rules.
- Multiple definitions can be written as a single rule, with the different definitions separated by the symbol | meaning logical OR.
- A If statement in Java is described with the rules

<if_stmt> → if (<logic_expr>) <stmt>

<if_stmt> → if (<logic-expr>) <stmt> else <stmt>

or with a single rule

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

- In these rules <stmt> represents either a singe stmt or a compound statement.
- BNF can be used to describe nearly all of the syntax of programing languages including nested structures to any depth and even imply operator precedence and operator associativity.

Describing lists

- In BNF, a lists of syntactic elements is defined using recursion.
- A rule is recursive if its LHS appears in its RHS. The following rules illustrate how recursion is used to describe lists:

```
<ident_list> → identifier
             | identifier, <ident_list>
```

Grammars and derivations

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

Ex:-

For a given grammer

```
<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 is as 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.

Ex2:-

Grammer

<assign> → <id> = <expr>

<id> ⟶ A | B | C

<expr> → <id> + <expr>

| <id> * <expr>

| (<expr>)

| <id>

derive A = B * (A + C).

Ex3:-

E → I E → E+E E → E*E E → (E) I → a | b | c

a*b+C;

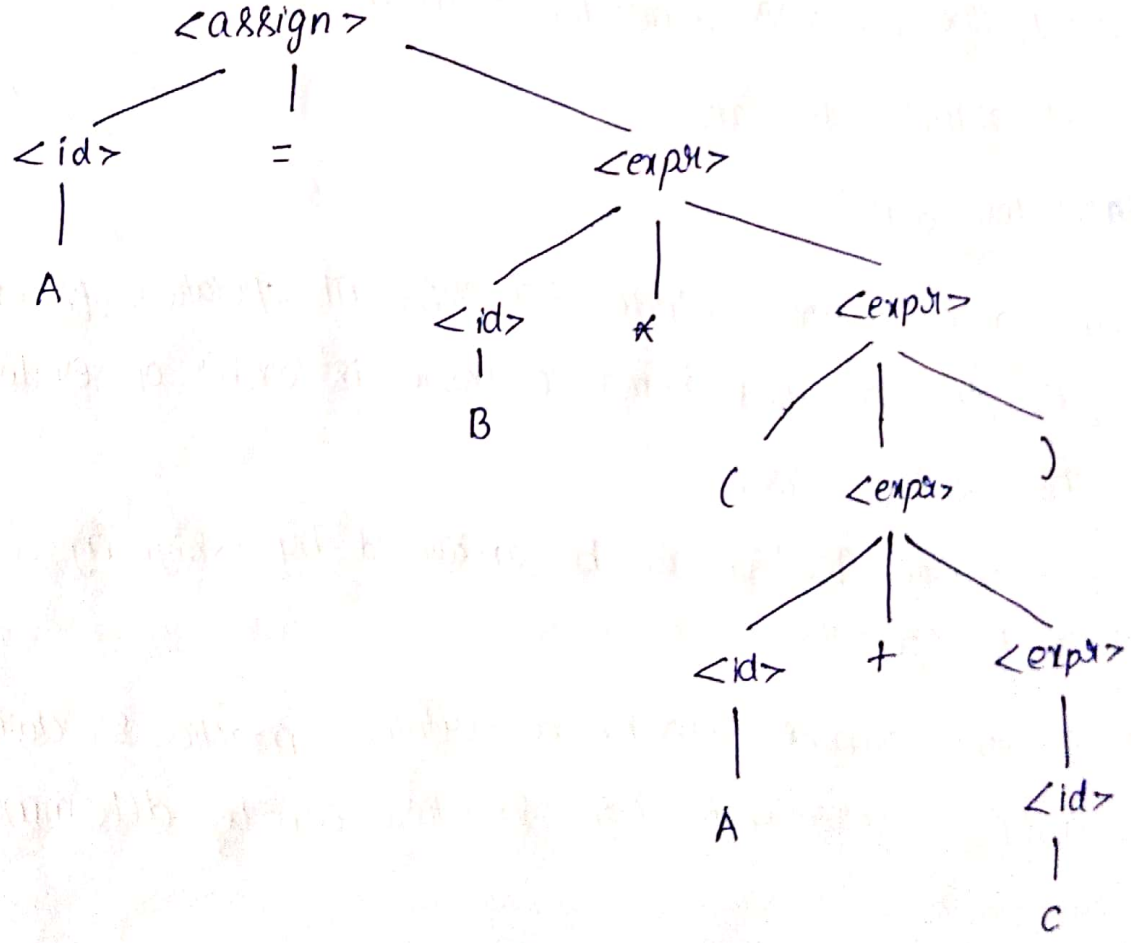
E → I
E → E*E
→ E* E+E
→ a*b+c

E → E+E
E → E*E+E
E*E+E

- In this derivation, the replaced nonterminal is always the leftmost non-terminal in the previous sentential form. Derivations that use this order of replacement are called leftmost derivations.

Parse trees

- One of the most attractive features of grammers 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 structure of the previous example is as follows:



- 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.

Ambiguity

- A grammer which generates two or more distinct parse trees is said to be ambiguous.
- If a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely.
- There are several other characteristics of a grammer that are sometimes useful in determining whether a grammer is ambiguous. They include the following:
    1) If the grammer generates a sentence with more than one leftmost derivation.
    2) If the grammer generates a sentence with more than one rightmost derivation.

Operator precedence

- When an expression includes two different operators, for example, x+y*z, one obvious semantic issue is order of evaluation of the two operator.
- This semantic question can be answered by assigning different precedence levels to operators.
- A grammer can be describe a certain syntactic structure so that part of the meaning of the structure can be determined from its parse tree.
- The operator that is generated lower in the parse tree can be used to indicate that it has precedence over an operator produced higher up in the tree.

A grammer can be written for the simple expressions that is both unambiguous and specifies consistent precedence of + and * operators, regardless of the order in which the operators appear in an expression.

The correct ordering is specified by using separate nonterminal symbols to represent the operands of the operators that have different precedence.

This requires additional non terminals and some new rules. So

So the previous grammer is rewritten as follows.

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

⇒

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term>
| <term>

<term> → <term> * <factor>
| <factor>

<factor> → (<expr>)
| <id>

Associativity of operators

When an expression includes two operators that have the same precedence we use the semantic rule called associativity.

A grammer for expressions may correctly imply operator associativity. When a grammer rule has its LHS also appearing at the begining of its RHS, the rule is said to be left recursive. This left recursion specifies left associativity.

- In some syntax algorithms, left recursion is not allowed. In such cases, the grammer must be modified to remove the left recursion.
- In most languages, right associativity occurs. So for this right recursion can be used.
- A grammer rule is right recursive if the LHS appears at the right end of the RHS.
    - Rule as the following one can be used to describe exponentiation

<factor> → <exp> ^ <factor>
| <exp>
<exp> → (<expr>)
| id

An unambiguous grammer for if-then-else

- The BNF rule for an if-then-else statement is as follows

<if_stmt> → if <logic_expr> then <stmt>
| if <logic_expr> then <stmt> else <stmt>

- If we also have a <stmt> → <if_stmt>, this grammer is am biguous.
- The simplest form that illustrates this ambiguity is:

if <logic_expr> then if <logic_expr> then <stmt> else <stmt>

example
```
if (done == true)
    then if (denom == 0)
        then quotient = 0
        else quotient = num / denom;
```

- The problem is that ~~if the~~ for the above grammer, if there is no matching else statement for every if statement, then the grammer is ambiguous. The problem is referred to as dangling else problem.
- The problem of the grammer is that it treats all statements as if they had equal syntactic significance.
- To reflect the difference, different abstractions or non terminals must be used.

<Stmt> → <matched> | <unmatched>

<matched> → if <logic_expr> then <matched> else <matched>
| any non-if statement.

<unmatched> → if <logic_expr> then <stmt>
| if <logic_expr> then <matched> else <unmatched>

Context-free grammar

A context-free grammar is a quadruple (NT, T, R, S), where

- NT is a finite set of symbols called Non-terminal symbols.
- T is a finite set of symbols called terminals.
- R is a finite set of productions or rules.
- S is the start symbol

Ex:- $P \to a$

$P \to b$

$P \to aPa$

$P \to bPa$

CFG = {P}, {a, b}, R, P}.

Extended BNF

- Because of a few minor inconveniences in BNF, it has been extended in several ways. These are called Extended BNF or EBNF.
- The extensions does not enhance the descriptive power of BNF, they only increase its readability and writability.
- Three extensions are commonly included in the various versions of EBNF.

1) An optional part in RHS.

Ex: if_else statement is described as

<if_Stmt> → if (<expression>) <Statement> [else <statement>]

2) The use of braces in RHS to indicate that the enclosed part can be repeated indefinitely or left out altogether.
   - This extension allows lists to be built with a single rule, instead of using recursion and two rules.

<ident_list> → <identifier> {, <identifier>}

3) The third extension deals with multiple-choice options. When a single element must be chosen from a group, the options are placed in parantheis and separated by OR operator.

<term> → <term> * <factor>
| <term> / <factor>
| <term> % <factor> is replaced by

<term> → <term> (*|/|%) <factor>.

- The brackets, braces and parantheses in the EBNF are metaSymbols.

Ex:-

$A \rightarrow A + B \mid A - B \mid B$

$B \rightarrow B * C \mid B / C \mid C$

$C \rightarrow A ** C \mid A$

$A \rightarrow (A) \mid id$

$id \rightarrow a \mid b \mid c$

EBNF $\Rightarrow$

$A \rightarrow A (+|-) B \mid B$

$B \rightarrow B (*|/) C \mid C$

$C \rightarrow A ** C \mid A$ $\Rightarrow$

$A \rightarrow (A) \mid id$

$id \rightarrow a \mid b \mid c$

$\Downarrow$

$A \rightarrow B \{ (+|-) B \}$

$B \rightarrow C \{ (*|/) C \}$

$C \rightarrow A \{ ** A \}$

$A \rightarrow (A) \mid id$

$id \rightarrow a \mid b \mid c$

Attribute grammars

Attribute grammars are context-free grammars to which attributes, attribute computation functions and predicate functions are added.

Attributes → grammar symbols (terminals and non terminals), which are similar to variables to which values are assigned.

Attribute computation functions → Sometimes called semantic functions, which specify how attribute values are computed.

predicate functions → These state the static semantic rules of a language.

Grammer

⇒ An attribute grammar is a grammar with the following additional features:

1) Associated with each grammar symbol x is a set of attributes A(x). The set A(x) has two disjoint sets S(x) and I(x)

S(x) → Synthesized attributes - which are used to pass semantic information up a parse tree

I(x) → pass semantic information down and across a tree.

2) Associated with each grammar rule is a set of semantic functions and a possible empty set of predicate functions over the attributes of the symbols in the grammer rule.

Ex:- For a rule $X_0 \rightarrow X_1, X_2, \ldots, X_n$

- Synthesized attributes of X_0 are computed with semantic functions

$S(X_0) = f(A(X_1), \ldots, A(X_n))$

Attribute grammars / context-sensitive grammars

- An attribute grammar is a device used to describe more of the structure of a programming language that can be described with a context-free grammar.
- Attribute grammar is an extension to a context-free grammar, as it can describe more languages.

Static Semantics

There are some characteristics of the structure of a programming languages that are difficult to be described with BNF

Ex:- 1) Compatibility rules :- A floating-point value cannot be assigned to an integer-type variables. To implement this restriction. additional grammer rules are needed and to implement all these typing rules, in BNF, the grammer would become too large.
- Size of grammer determines the size of syntax analyzer.

2) All variables have to be declared before they are referenced. This rule also cannot be specified in BNF.

- The above examples are the categories of language rules called static semantics.
- The static semantics of a language only indirectly related to the meaning of programs during execution. It is named as static semantics because the analysis required to check these specifications can be done at compile time.
- Attribute grammers are formal approach for describing and checking the correctness of the static semantic rules of a program.

The predicate rule states that the name string attribute of <proc_name> nonterminal in the procedure must match the name string attribut of procedure following the end of the procedure.

Example 2 :- Attribute grammar to check type rules in assignment statements

- Three variables A, B and C. The Right side of assignments can be either a variable or an expression, in the form of variable addition. There can be mixed type of operands on right side, but the assignment is valid only if the RHS expression and LHS type are same.

<assign> → <var> = <expr>

<expr> → <var> + <var>

| <var>

<var> → A | B | C

- So the each grammar symbol has two attributes :

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 case of variable, the actual type is intrinsic, but for an expression, it is determined by actual type of child nodes.

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

- Values of the synthesized attribute on a parse tree depend only on the values of the attributes on the node's children nodes.

→ Inherited attributes of symbols X_j, $1 \le j \le n$ are computed with semantic function of the form

$$I(X_j) = f(A(X_0), \ldots A(X_n))$$

- The value of an inherited attribute depends on node's parent node and those of its sibiling nodes.

→ A predicate function has the form of a boolean expression on the union of the attribute set $\{A(X_0), \ldots, A(X_n)\}$ and a set of literal attribute values.

Intrinsic 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.

Examples of attribute grammers

1) - Consider the following fragment of an attribute grammar that describes that the name of the end of an "Ada" procedure must match the procedure's name. [This rule cannot be stated in BNF).

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

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

- The complete attribute grammer is as follows:

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>.actualtype ← <var>.actual_type

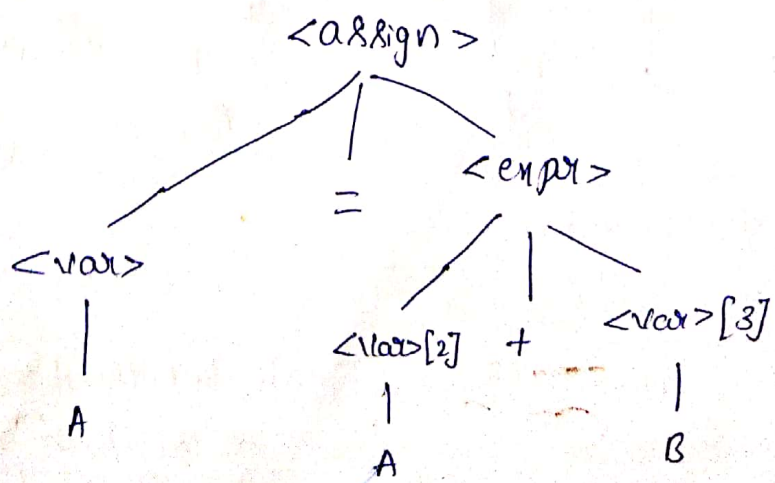
   predicate: <expr>.actualtype == <expr>.expected_type

4. Syntax rule: <var> → A|B|C

   Semantic rule: <var>.actual_type ← lookup (<var>.string).

- The look-up function looks up for a given variable name in the symbol table and returns the variable's type.

A = A+B parse tree



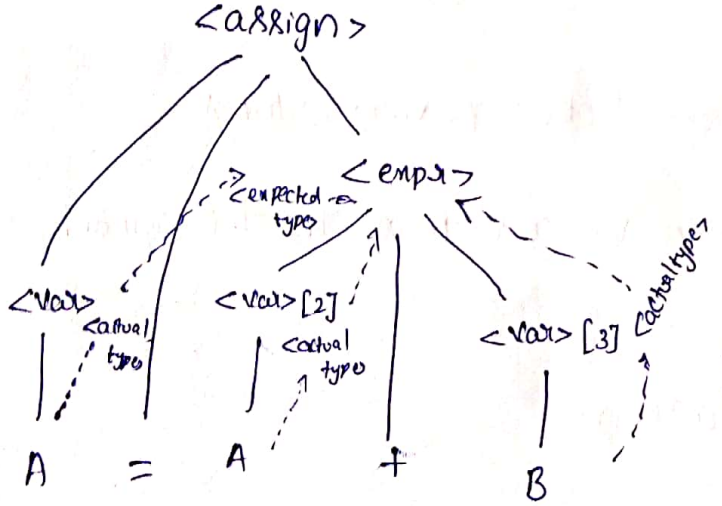
Computing attribute values

- process of computing the attribute values of a parse tree is sometimes called decorating the parse tree.
- If all attributes are inherited, this process could proceed in top-down order, while if all attributes are synthesized, it proceeds in bottom-up fashion.
- Our grammer has both types. The order of evaluation is as follows.

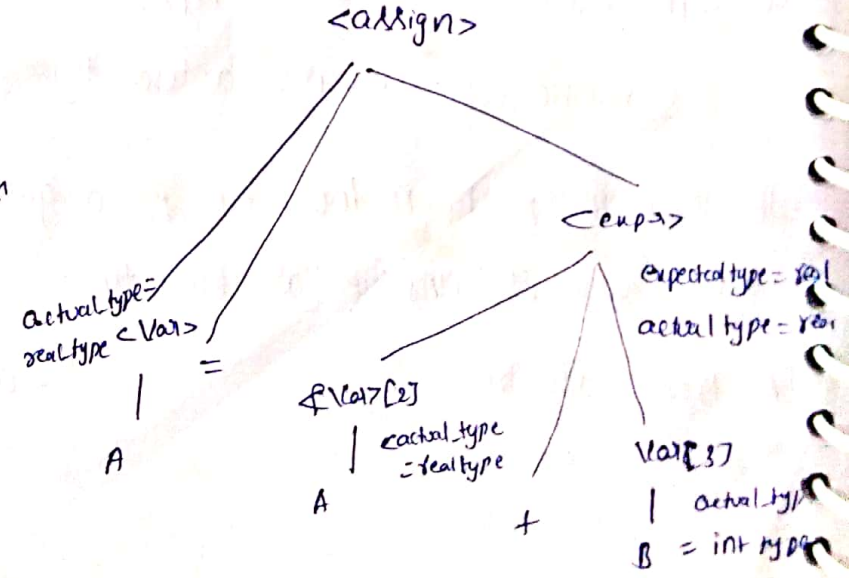
1. <var>.actual_type ← look-up(A) - Rule 4
2. <expr>.expected_type ← <var>.actual_type - Rule 1
3. <var>[2].actual_type ← look-up(A) - Rule 4

   <var>[3].actual_type ← look-up(B) - Rule 4
4. <expr>.actual_type ← either int or real - Rule 2
5. <expr>.expected_type == <expr>.actual_type is either TRUE or FALSE - Rule 2

Parse tree



Full attributed parse tree



- Solid lines show parse tree
- dashed lines show attribute flow.

→ A grammar is called S-attributed if all attributes are synthesized.

→ A grammar is called L-attributed if the parse tree traversal is left-to-right and depth-first.

Describing Dynamic Semantics

- Semantics refers to the meaning of expressions, statements and program units of a programming language.
- There is no universally accepted notation or approach for dynamic semantics. But there are several methods for describing semantics:
  1) operational Semantics
  2) Denotational Semantics
  3) ~~Axiomatic Semantics~~

operational Semantics

- operational semantics are used to describe the meaning of a statement or a program by specifying the effects of running it on a machine.
- The effects on the machine are viewed as the sequence of changes in its state, where the machine's state is the collection of the values in its storage.
- There are different levels of operational semantics.
  1) Natural operational semantics
     - Determining the meaning of the program, after the final result state.
  2) Structural operational Semantics
     - Determining the meaning of the program through an examination of the complete sequence of state changes that occur when the program is executed.

⇒ The first step in creating an operational semantics description of a language is to design an appropriate intermediate language.

- For example, the semantics of the C for construct can be described in simple statements is as follows:

C Statement	Meaning

for (expr1 ; expr2 ; expr3)
{
stmts;
}
stmts.
	expr 1;
	loop: if expr2 == 0 goto out
	stmts
	expr3;
	goto loop
	out : stmts.

- The intermediate language is meant to be convienent for machine rather than human readers.

- A simple intermediate language will have statements in the following manner:

ident = var bin_op var

ident = un_op var.

where bin_op is a binary arithmetic operator

un_op → unary operator.

ident → identifier

var → variable.

- operational semantics depends on programming languages of lower levels. The statements of one programming language are described in terms of the statements of a lower-level language.

- Denotational and axiomatic semantics are based on mathematics and logic and are more formal compared to operational semantics.

Denotational Semantics

- It is most widely known formal method for describing the meaning of programs.
- The process of constructing a denotational semantics specification for a programming language requires one to define each language entity as an object mathematical object and a function that maps instances of that language entity onto instances of the mathematical object.
- The difficulty with this method lies in creating the objects and the mapping functions.
- The method is named denotational because, the mathematical objects denote the meaning of their corresponding Syntactic entities.
- In operational semantics, programming language constructs are translated into simpler programming language constructs where as in denotational semantics, programming language constructs are mapped to mathematical objects and functions.
- unlike operational semantics, denotational semantics do not model the step-by-step computational proceeding of programs.

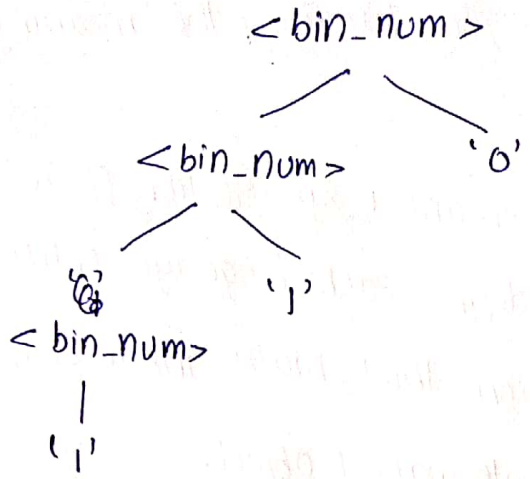
Example

⇒ character string representations of binary numbers

- The syntax for such representation is as follows:

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

- A parse tree for the binary number 110 is as follows.



⇒ we place quotes around digits to show that they are syntactic digits not mathematical digits.

- The syntactic domain of the mapping function for binary numbers is the set of all character string representations of binary numbers. The semantic domain is the set of non-negative decimal numbers symbolized by N.

- The semantic function named M_{bin} maps the syntactic objects as described in the previous grammer rules to the objects in N. The function M_{bin} is defined as follows:

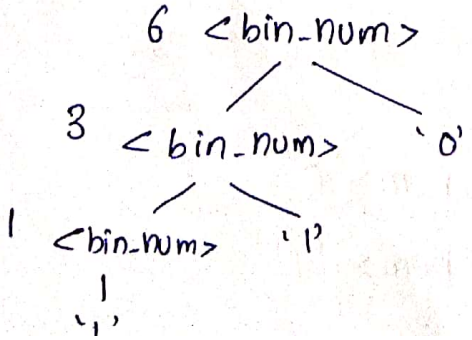
$$M_{bin}('0') = 0$$

$$M_{bin}('1') = 1$$

$$M_{bin}(<bin\_num>\ '0') = 2 * M_{bin}(<bin\_num>)$$

$$M_{bin}(<bin\_num>\ '1') = 2 * M_{bin}(<bin\_num>) + 1$$

- The meanings or denoted objects can be attached to the nodes of the parse tree. This is syntax-directed semantics.



⇒ Convert 10110 to a decimal number

$M_{bin}("10110") = 2 * M_{bin}("1011")$

$= 2 * [2 * M_{bin}("101") + 1]$

$= 2 * \{2 * [2 * M_{bin}("10") + 1] + 1\}$

$= 2 * \{2 * [2 * (2 * M_{bin}("1")) + 1] + 1\}$

$= 2 * \{2 * [2 * (2 * 1) + 1] + 1\}$

$= 2 * \{2 * [2 * 2 + 1] + 1\}$

$= 2 * \{2 * [5] + \}$

$= 2 * 11 = 22$,,

Example 2 - Describing semantic meaning of syntactic decimal literals.

- In this case the syntactic domain is the set of character representations of decimal numbers and the semantic domain is again N.

<dec_num> ⟹ '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

| <dec_num> ('0' | ... | '9')

- The denotational mapping for these syntax rules are:

$M_{dec}('0') = 0$, $M_{dec}('1') = 1$, ..., $M_{dec}('9') = 9$.

$M_{dec}(<dec\_num>\ '0') = 10 * M_{dec}(<dec\_num>)$

$M_{dec}(<dec\_num>\ '1') = 10 * M_{dec}(<dec\_num>) + 1$

⋮

$M_{dec}(<dec\_num>\ '9') = 10 * M_{dec}(<dec\_num>) + 9$.

Lexical analysis and Syntax analysis.

- There are three approaches to implementing programming languages:
    1) compilation
    2) pure interpretation.
    3) Hybrid

Compilation

- This approach uses a program called compiler, which translates programs written in high-level language programming language into machine code.

    Ex: C, C++, COBOL.

- These are used to implement larger applications.

pure interpretation

- These systems perform no translation.
- programs are interpreted in their original form by a software interpreter.
- pure interpretation is usually used for smaller systems.

    Ex: Python, JavaScript.

Hybrid

- These implementations translate programs written in high-level languag into intermediate form, which is latter interpreted.

    Ex: Java.

⇒ All the three implementation approaches use both lexical and Syntax analyzers.

- Nearly all compilers separate the task of analyzing syntax into two distinct parts named lexical analysis and Syntax analysis.
- Lexical analysis deals with small-scale language constructs such as names and numeric literals, while the syntax analysis 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:

1) Simplicity

- techniques for lexical analysis are less complex than those required for syntax analysis; Removing the low-level details of lexical analysis from the syntax analyzer make the syntax analyzer both smaller and less complex.

2) Efficiency

- Lexical analysis requires a significant portion of total compilation time. So it pays to optimize the lexical analyzer.

3) portability

- Lexical analyzer is somewhat platform dependent. However the syntax analyzer can be platform independent. So it is good to isolate machine-dependent parts of any software system.

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.

- 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.
- The logical groupings are named Lexemes and the internal codes for categories of these groupings are named tokens.

Ex:- result = oldsum-value/100;

Token	lexeme
Ident	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
Div_OP	/
INT_LIT	100
SEMICOLON	;

- Each call to the lexical analyzer returns a single lexeme and its token.
- The lexical-analysis process includes skipping comments and white spaces outside lexemes, as they are not relevant to the meaning of the program.
- Also lexical analyzer inserts lexemes for user-defined names into the symbol table, which is used by later phases of compiler.
- Lexical analyzers detects ~~correct~~ syntactic errors in tokens and report such errors to the user.

- There are three approaches to building a lexical analyzer:

1) Write a formal description of the token patterns of the language using a descriptive language related to regular expressions. These descriptions are used as input to a software tool that automatically generates a lexical analyzer.

   Ex:- lex in UNIX systems.

2) Design a state transition diagram that describe the token patterns of the language and write a program that implements the diagram.

3) Design a state transition diagram that describe the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

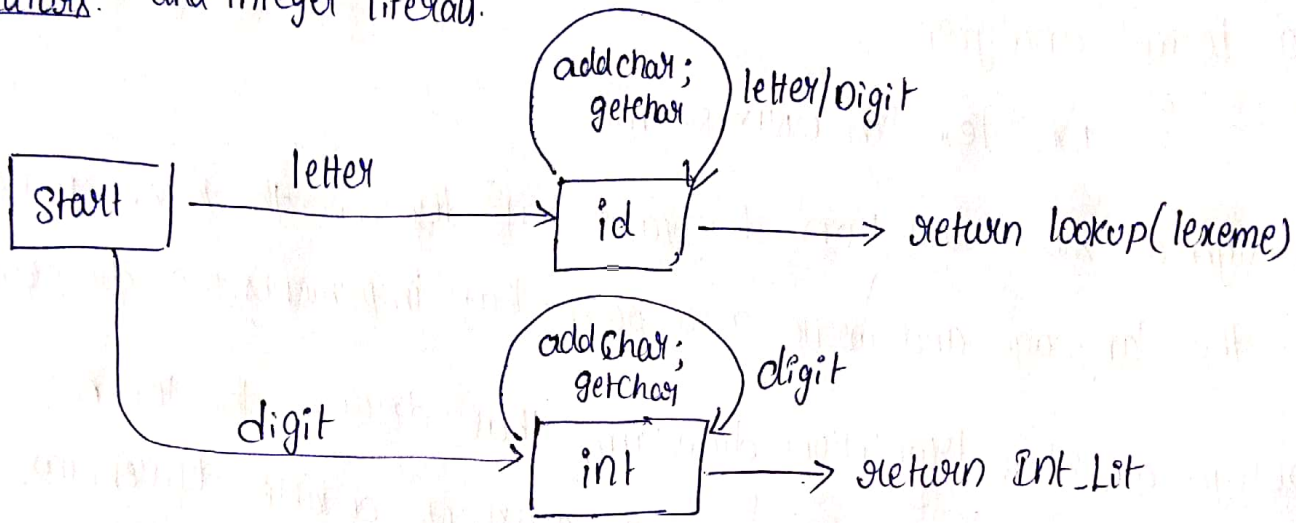
⇒ A state transition diagram or just state diagram is a directed graph. The nodes of a state diagram are labeled with state names. The arcs are labeled with the input characters that cause the transitions among the states.

- State diagrams of the form used for lexical analyzers are representations of a class of mathematical machines called finite automata.
  - Finite automata can be designed to recognize members of a class of languages called regular languages.
  - Regular grammars are generative devices for regular languages.

- The tokens of a programming language are a regular language and a lexical analyzer is a finite automata.

Example

- Suppose we need a lexical analyzer that recognizes only ~~arithmetic expressions~~, including variable names and integer literals as operands.
- The state diagram to recognize names, ~~parentheses and arithmetic operators~~. and integer literals.



Parsing

- The process of analyzing syntax that is often referred to as Syntax analysis is usually called as parsing.
- parsers for programming languages construct parse trees for given programs.
- Both parse trees and derivations are the required syntactic information needed by a language processor.

- The two main goals of Syntax analysis are:

  1) It 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 continue the analysis of remaining statements in the program.

2) The second goal of syntax analysis to produce a complete parse tree. This is used as a base for translation.

→ parsers are categorized according to the direction in which they build parse trees. The two broad classes of parsers are "top-down", in which tree is built from the root downwards to the leaves and "bottom-up", in which the parse tree is built from leaves upward to the root.

Topdown parsers

A top-down parser traces or builds a parse tree in preorder. It corresponds to a left-most derivation.

In this, the parser must choose among the rules, which helps in deriving the required string. This is the parsing decision problem for top-down parsers.

The most common top-down parsers choose the correct RHS for the leftmost nonterminal, by comparing the next token of input, with the first symbols that can be generated by the RHS's of the rules.

Example

$E \rightarrow I$

$E \rightarrow E+E$

$\quad | \; E*E$

$\quad | \; (E)$

$I \rightarrow a \mid b \mid c$

$(a*b+c)$

$E \rightarrow (E)$

$\rightarrow (E*E)$

$\rightarrow (I*E)$

$\rightarrow (a*E)$

$\rightarrow (a*E+E)$

$\rightarrow (a*I+E)$

$\rightarrow (a*b+E)$

$\rightarrow (a*b+I)$

$\rightarrow (a*b+c)$.

- The common top-down parsing algorithms is recursive-descent parser, which is a coded version of syntax analyzer based on the BNF descriptions.
- The other alternative is to use a parsing table rather than code to implement the BNF rules.
- Both these algorithms are called LL algorithms. The first L in LL specifies a left to right scan of input and the second 'L' specifies left most derivation is generated.

Bottom-up parser

- A bottom-up parser constructs a parse tree by begining at the leaves and progressing towards the root. This corresponds to reverse of Right most derivation.
- In this bottom-up parsing, the decision problem is to determine which substring of the initial given sentence is the RHS to be reduced to its corresponding LHS. The choosen RHS's among the multiple options is called handle.

Example

$S \rightarrow aAc$

$A \rightarrow aA \mid b$

required sentence → aabc

a a b c

⇓

a a A c

⇓

a A c

⇓

S

Recursive-Descent parsing

- A recursive-descent parser is a kind of top-down parser built from a set of recursive procedures.
- In this each procedure usually implements one of the productions or rules of a grammar.
- It recursively parses the input to make a parse tree, which may or may not require back tracking.
- A form of recursive-descent parsing that does not require any backtracking is known as predictive parsing.

predictive-parser

- A predictive parser has the capability to predict which production is to be used to replace the input string.
- It does not suffer from backtracking.
- To accomplish its task, it uses a look-ahead pointer, which points to the next input symbol.
- It has a subprogram for each non terminal of its associated grammar.
- The responsibility of the subprogram is:

  when given an input string, it traces out the parse trees that can be rooted at the nonterminal and whose leaves match the input string.

Example

<expr> → <term> { (+|-) <term> }

<term> → <factor> { (*|/) <factor> }.

<factor> → id | int_const | (expr>)

⇒ The recursive descent parser has three functions expr, term and factor.