Principles of Programming Languages

Unit 1

Third Semester
MScCSIT, TU

Lalita Sthapit

Unit 1:

1.1 Introductory Concepts

- Reason for studying concept of Programming Languages
- History of the development of programming language
- Programming Language Domains
- Attributes of Programming Language
- Standardization
- Programming Environments

Why study programming languages?

- To improve your ability to develop effective algorithms
- To improve your use of existing programming languages
- To increase your vocabulary of useful programming constructs
- To allow a better choice of programming language
- To make it easier to learn a new language
- To make it easier to design a new language

History of the development of programming language

On the basis of

- Language development with respect to application (domain)
- Software architecture

Other terms:

- Programming language domains
- Programming paradigm (steps to implement)
  - Imperative
  - Applicative
  - Rule based
  - Object Oriented

- Syntax and Semantics

Attributes of good programming language

- Conceptual integrity: Clarity, simplicity, and unity - provides both a framework for thinking about algorithms and a means of expressing those algorithms
- Orthogonality -every combination of features is meaningful. Some negative aspects
- Naturalness for the application - program structure reflects the logical structure of algorithm
- Support for abstraction - program data reflects problem being solved
- Ease of program verification - verifying that program correctly performs its required function
- Programming environment - external support for the language: editors, testing programs.
- Portability of programs - transportability of the resulting programs from the computer on which they are developed to other computer systems
- Cost of use - program execution, program translation, program creation, and program maintenance

Language Standardization

- The need for standards is to increase portability of programs. Language standards are defined by national standards bodies:
  - ISO - International Standards organization
  - IEEE - Institute of Electrical and Electronics Engineers
  - ANSI - American National Standards Institute

- Standardization comes in two flavours
  - Proprietary standards
  - Consensus standards

- All work in a similar way:
  - Working group of volunteers set up to define standard
  - Agree on features for new standard
  - Vote on standard, disagreements are worked out
  - If approved by working group, submitted to parent organization for approval.
  - It may be partially technical and partially political

Standardization contd..

- **Problems:** When to standardize a language?

  - If too late - many incompatible versions - FORTRAN in 1960s was already a de facto standard, but no two were the same
  - If too early - no experience with language - Ada in 1983 had no running compilers
  - What happens with the software is developed before the standardization?
  - Ideally, new standards have to be compatible with older standards.

To use standards effectively, we need to address three issues

1. Timeliness (decide when to standardize a language)

2. Conformance (conformant program, conformant compiler, it does not say anything about the standard extension)

3. Obsolescence (obsolescent features and deprecated features)

Internationalization

- Local conventions affect the way data are stored and processed.
- How to specify languages useful globally?

Some of the internationalization issues:

- What **character codes** to use/ ideographic systems
- **Collating** sequences? - How do you alphabetize various languages?
  - Sort
  - Case
  - Scanning direction
- **Dates**? - What date is 10/12/01? Is it a date in October or December?

- **Time**? - How do you handle time zones, summer time in Europe, daylight savings time in US, Southern hemisphere is 6 months out of phase with northern hemisphere, Date to change from summer to standard time is not consistent.

- **Currency**? - How to handle dollars, pounds, marks, francs, euros, etc

Programming environments

- **Programming environment**: consists of primary set of tools and a command language for invoking them, used as an aid during different stages of program development

  - Editor
  - Compilation / debugging
  - Execution
  - Testing (execution tracer features, breakpoints, assertions)
  - verifiers/ test data generators
  - printers

- Other Necessary language features:

  - Modular organization
  - Local/global variables
  - Libraries

Unit 1 contd ..

1.2 Language Design Issue

- Different Architectures
- Virtual Machine
- Binding Times

Different Architectures

The Operation of a computer

- **Computer:** An integrated set of algorithms and data structures capable of storing and executing programs.
- **Components:**
  - Data - types of elementary data items and data structures to be manipulated
  - Primitive Operations - operations to manipulate data
  - Sequence Control - mechanisms for controlling the sequence of operations
  - Data Access - mechanisms to supply data to the operations
  - Storage Management - mechanisms for memory allocation
  - Operating Environment - mechanisms for I/O , communication with external environment

Implementation of the components

1. Hardware
2. Firmware
3. Software
    a. Translation
    b. Software simulation
    c. Virtual machines
    d. Binding and binding times

1. **Hardware**
    a. CPU (Central processing Unit)
        Inboard memory
    b. I/O devices - keyboard, mouse, disk drives, etc
        Outboard storage - disks, CDs
    c. Hardware level operations

    a. CPU
        - ALU (arithmetic-logic unit)
        - Control Unit
        - Registers (fast memory)

- PC - program counter, location counter, program address register, instruction counter
- IR - instruction register

Hardware contd ..

b. Inboard memory

- Registers (listed above)
- Cache memory
- Main memory

c. Hardware-level operations

- Fetch-execute cycle:
  Processor **fetches** an instruction from memory.
  Processor **executes** the fetched instruction (performs the instruction cycle)

  Steps in fetch-execute cycle:
  - Program counter- contains the address of the next instruction to be fetched.
  - The fetched instruction is loaded into Instruction Register
  - Program counter is incremented after fetching.
  - Processor interprets the bits stored in the IR and performs the required action.
- Instruction cycle - processing that is required for a single instruction.

The instructions executed by CPU are machine language instructions. Some of them are implemented in hardware - e.g. increment a register by 1. Some are implemented in Firmware - e.g. add the contents of two registers.

2. Firmware

- Firmware is a set of machine-language instructions implemented by programs, called *microprograms*,
  stored in read-only memory in the computer (ROM).

- Advantages:
  - flexibility - by replacing the ROM component we can increase the set of machine instructions,

  - less cost of the hardware - the simpler the instructions, the easier to hardwire them

3. Software

1. Translators and virtual architectures

Translation (compilation):

Input: high-level language program
Output: machine language code

Types of translators (high to low level):

Preprocessor:

Input: extended form of high-level language
Output: standard form of high-level language

Compiler:

Input: high-level language program
Output (object language): assembler code

Assembler:

Input: assembly language
Output: one-to-one correspondence to a machine language code

Loader/Link editor:

Input: assembler/machine language relocatable program
Output: executable program (absolute addresses are assigned)

3. Software simulation - Interpreters

The program is not translated to machine language code.
Instead, it is executed by another program.

Example: Prolog interpreter written in C++

Advantages of interpreted languages:

- very easy to be implemented
- easy to debug
- flexibility of language - easy to modify the interpreter
- portability - as long as the interpreter is portable, the language is also portable.
- Disadvantages of interpreted languages: slow execution.

3. Virtual Machines

Program: data + operations on these data

Computer: implementation of data structures + implementation of operations

- **Hardware computer:** elementary data items, very simple operations
  Firmware computer: elementary data items, machine language instructions

Software computer: each programming environment defines a specific software computer.

eg. - the operating systems is one specific virtual computer
A programming language also defines a virtual computer.

- **Basic Hierarchy:**

  Software
  Firmware
  Hardware

Software sub-hierarchy - depends on how a given programming environment is implemented

Eg. of virtual machines hierarchy:

Java applets

Java virtual machine - used to implement the Java applets

C virtual machine - used to implement Java

Operating system - used to implement C

Firmware - used to implement machine language

Hardware - used to implement firmware microprograms

4. Binding and Binding Times

- Binding - fixing a feature to have a specific value among a set of possible values.

  eg. - your program may be named in different ways and when you choose a particular name you have done a binding.

- Different programming features have different binding times, depending on:
  - the nature of the feature, e.g. you choose the names of the variables in the source code, the operating system chooses the physical address of the variables.

  - the implementation of the feature - in certain cases the programmer has a choice to specify the binding time for a given feature.

- **Binding occurs at:**
  - **At language definition** - concerns available data types and language structures,
    e.g in C++ the assignment statement is **=**, while in Pascal it is **:=**
  - **At language implementation** - concerns representation of data structures and operations,
    e.g. representation of numbers and arithmetic operations

Binding times contd ..

- **At translation** -

  - Chosen by the programmer - variable types and assignments
  - Chosen by the compiler - relative locations of variables and arrays
  - Chosen by the loader - absolute locations

- **At execution** -

  - Memory contents
  - On entry to a subprogram (copying arguments to parameter locations)At arbitrary points (when executing assignment statements)
  - Recursive programs and dynamic libraries

- **Example:** X = X+10 (refer pg.no. 62 chapter 2 from text book)

- **Importance of binding times**

  - If done at translation time - more efficiency is gained

  - If done at execution time - more flexibility is gained.

Unit 1 contd..

1.3. Translation Issues

- Syntatic criteria, Syntatic elements
- Formal methods of describing syntax – BNF, RG, Parse tree, Ambiguity, Parsing Algorithm
- Semantic modeling – attribute grammar, denotational semantics.

Programming Language Syntax

Syntax of a programming language describes the structure of programs without any consideration of their meaning.

Examples of syntax features:

- statements end with ';' (C,C++, Pascal), with'.' (Prolog), or do not have an ending symbol (FORTRAN)
- variables must start with any letter (C, C++, Java), or only with a capital letter (Prolog).
- the symbol for assignment statement is '=', or ':=' , or something else.

Key criteria concerning syntax

- Readability – a program is considered readable if the algorithm and data are apparent by inspection.
- Write ability – ease of writing the program.
- Verifiability – ability to prove program correctness (very difficult issue)
- Translatability – ease of translating the program into executable form.
- Lack of ambiguity – the syntax should provide for ease of avoiding ambiguous structures.

Basic syntactic concepts in a programming language

- Character set – the alphabet of the language.
  Several different character sets are used: ASCII, EBCIDIC, Unicode.
- Identifiers – strings of letters of digits usually beginning with a letter
- Operator Symbols – + - * /
- Keywords or Reserved Words – used as a fixed part of the syntax of a statement.
- Noise words – optional words inserted into statements to improve readability.
- Comments – used to improve readability and for documentation purposes. Comments are usually enclosed by special markers.
- Blanks – rules vary from language to language. Usually only significant in literal strings.
- Delimiters – used to denote the beginning and the end of syntactic constructs.
- Expressions – functions that access data objects in a program and return a value
- Statements – these are the sentences of the language, describe a task to be performed.

Overall Program-Subprogram Structure

- **Separate subprogram definitions:** separate compilation, linked at load time. Advantages: easy modification.
- **Separate data definitions:** Group together all definitions that manipulate a data object.
  General approach in OOP.
- **Nested subprogram definitions:** Subprogram definitions appear as declarations within the main program or other subprograms. Not used in many contemporary languages. Provides for static type checking in non-local referencing environments.
- **Separate interface definitions:** Subprogram interface - the way programs and subprograms interact by means of arguments and returned results. A program specification component may be used to describe the type of information transferred between separate components of the program. E.G. C/C++ use header files as specification components.
- **Data descriptions separated from executable statements**. A centralized data division contains all data declarations. E.G. COBOL. Advantage - logical data format independent on algorithms.
- **Unseparated subprogram definitions**: No syntactic distinction between main program statements and subprogram statements. Allows for run-time translation and execution.

Stages in Translation

- Lexical analysis (scanning) – identifying the tokens of the programming language.
- Syntactic analysis (parsing) – determines the *structure* of the program, as defined by the language *grammar*.
- Semantic analysis - assigns meaning to the syntactic structures, the semantic analysis builds the bridge between analysis and synthesis.
- Intermediate Code Generation
- Optimization
- Code generation

What is Bootstrapping?

Formal Translation Model

Syntax is concerned with the structure of programs. The formal description of the syntax of a language is called **grammar.**

- Grammars consist of rules and is used for both recognition and generation of sentences.
- Grammars are independent of the syntactic analysis.
- Language consists of sentences. Each sentence consists of words. The rules that tell us how to combine words that form correct sentences are called **grammar rules**.

  Example :

  "The plane flies." - VALID
  "Plane the flies" - INVALID

  The corresponding rule here says that the **article** must precede the **noun. "Article" and "Noun"** correspond to certain **categories of words** in the language. For example, the words *Plane, book, room, class*, are all **nouns**, while fly*, speak, write*, are **verbs**.

- **Why do we need word categories?**

  There are infinitely many sentences and we cannot write a rule for each individual sentence. We need these categories in order to describe the structural patterns of the sentences. For example, the basic sentence pattern in English is a **noun phrase** followed by a **verb phrase**.

  A sequence of words that constitutes a given category is called a **constituent**. For example, the boldface parts in each of the sentences below correspond to a constituent called **verb phrase**.

  The boy **is reading a book**.
  The boy **is reading an interesting book**.
  The boy **is reading a book by Mark Twain**.

Terminal and non-terminal symbols, grammar rules

- **How do we represent constituents and how do we represent words?**

- Grammars use two types of symbols:

  Terminal - to represent the words.
  Non-terminal - to represent categories of words and constituents.

  The symbols are used in **grammar rules**. Here are some examples:

Rule	**Meaning**
N → plane	**N** is the non-terminal symbol for "noun", "boy" is a terminal "symbol"
D → the | a | an	**D** is the non-terminal symbol for definite or indefinite articles.
NP → D N noun	this rule says that a noun phrase **NP** may consist of an article followed by a

- **There is one special non-terminal symbol S to represent "sentence". It is called also the starting symbol of the grammar.**
- **Grammars for programming languages - no major differences**

BNF notation

- Grammars for programming languages use a special notation called BNF (Backus-Naur form):
  - The non-terminal symbols are enclosed in < >.
  - Instead of → the symbol **::=** is used.
  - The vertical bar is used in the same way - meaning choice.
  - [] are used to represent optional constituents.
  - BNF notation is equivalent to the first notation in the examples above.

Example: The rule

<assignment statement> ::= <variable> = <arithmetic expression>

says that an assignment statement has a variable name on its left-hand side followed by the symbol "=", followed by an arithmetic expression.

Example: Expression syntax in **BNF notation:**

E → T

E → E + T <expression> ::= <term> | <expression> + <term>

T → P

T → T * P <term> ::= <primary> | <term> * <primary>

P → I

P → (E) <primary> ::=<integer> | (<expression>)

The only non-terninal symbol that remains undefined is **I**, meaning 'integer'. It is defined below using BNF notation:

<integer> ::= [-] <unsigned_integer>
<unsigned_integer> ::= <digit>
<unsigned_integer> ::=<digit><unsigned_integer>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

If we want to exclude 0 as a starting digit, we need to introduce two types of digits: non_zero_digit and any digit:

<unsigned_integer> ::= <non_zero_digit>
<unsigned_integer> ::= <unsigned_integer><any_digit>
<any_digit> ::= 0 | <non_zero_digit>
<non_zero_digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Extension to BNF Notation

- Extended version of BNF is called EBNF. The extension do not enhance the descriptive power of BNF but it only increases readability and writability. The properties of EBNF are
  - An optional element may be indicated by enclosing the element in square bracket []
  - A choice of alternatives may use the | symbol within a single rule, optionally enclosed by parentheses ([,]) if needed.
  - An arbitrary sequence of instances of an element may be indicated by enclosing the element in braces followed by an asterisk {…}*

  Eg.

  <signed integer>::= [+|-] <digit>{<digit>}*

  <identifier>::=<letter>{<letter>|<digit>}*

  Rule in BNF , <arithmetic expression> ::=<term>|<arithmetic expression>+<term>

  Can be represented in EBNF as :

  <arithmetic expression> ::=<term> {+<term>}*

BNF

1. <assignment statement> ::=<variable>=<arithmetic expression>
2. <arithmetic expression> ::=<term>|<arithmetic expression>+<term>| <arithmetic expression>-<term>|
3. <term>::=<primary>|<term>X<primary>|<term>/<primary>
4. <primary>::=<variable>|<number>|(<arithmetic expression>)
5. <variable>::=<identifier>|<identifier>[<subscript list>]
6. <subscript list>::=<arithmetic expression>|<subscript list>,<arithmetic expression>

Corresponding EBNF

1. <assignment statement> ::=<variable>=<arithmetic expression>
2. <arithmetic expression> ::=<term>{[+|-]<term>}*
3. <term>::=<primary>{[/|X]<primary>}*
4. <primary>::=<variable>|<number>|(<arithmetic expression>)
5. <variable>::=<identifier>|<identifier>[<subscript list>]
6. <subscript list>::=<arithmetic expression>{,<arithmetic expression>}*

CFG

expr → expr * term | expr / term | term

term → term + factor | term – factor |factor

Where V={expr, term, factor}, T= {*,/<+,-}

Concert to BNF and EBNF

BNF

<expr> ::= <expr> *< term> | <expr> /< term> |< term>

<term> ::= <term >+ <factor> | <term >– <factor> |<factor>

EBNF

<expr> ::= < term>{[*|/]< term>}*

<term> ::= <factor> {[+|-]<factor>}*

BNF

<compound>::=**begin**<stmt>{<stmt>}***end**

EBNF

<compound>::=**begin**{<stmt>}$^{+}$**end**

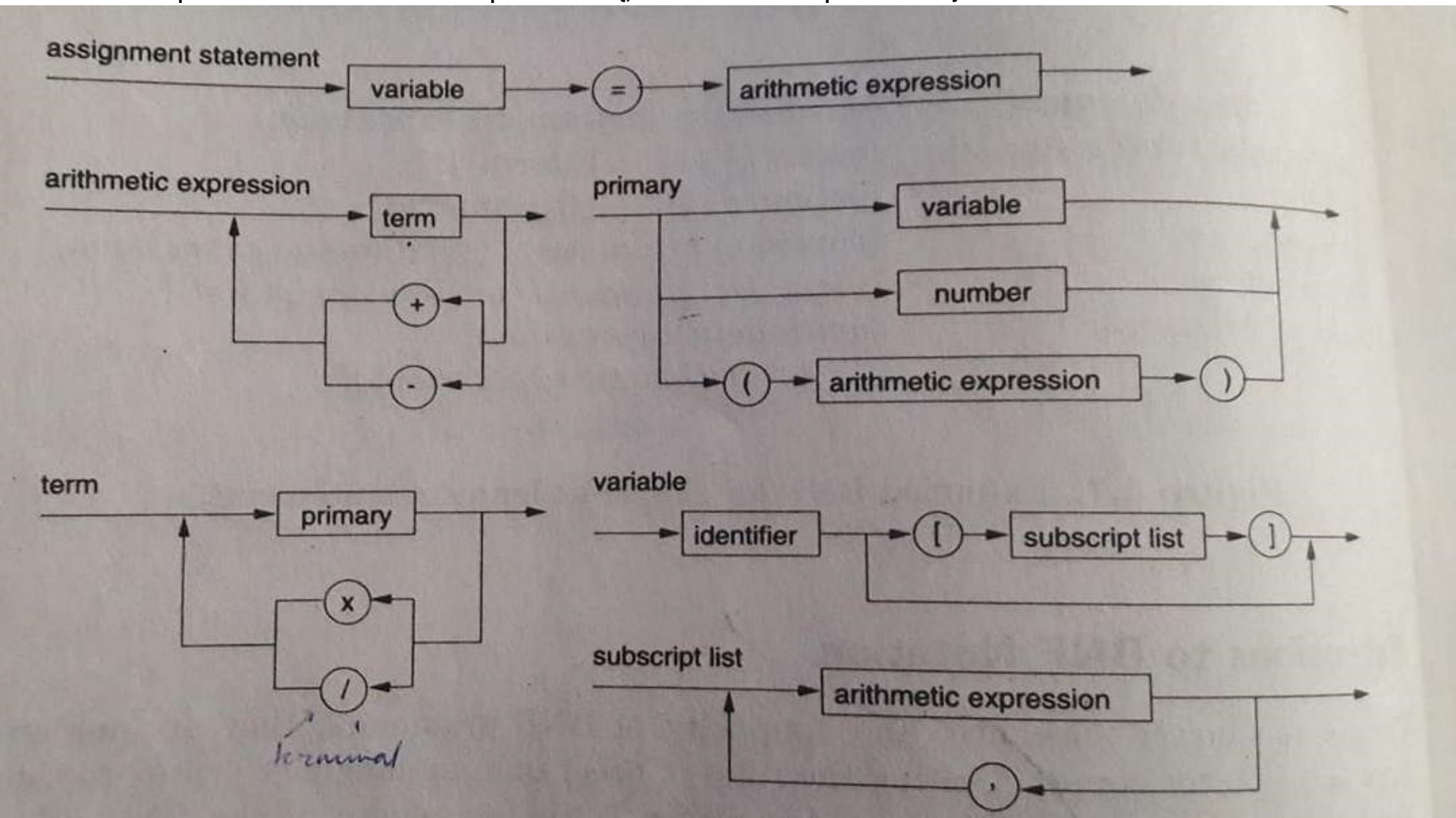
Convert to EBNF

<assign>::=<id>=<expr>
<id>::=A|B|C
<expr>::=<expr>+<expr>|<expr>*<expr>|<expr>|<id>

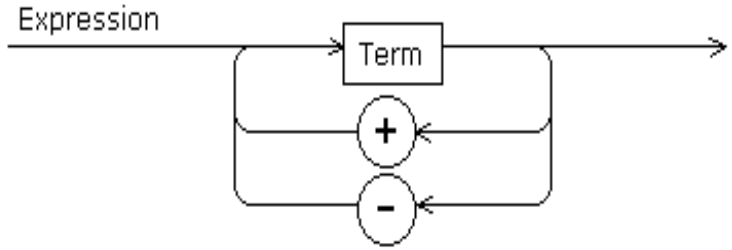
Syntax Chart/Graph

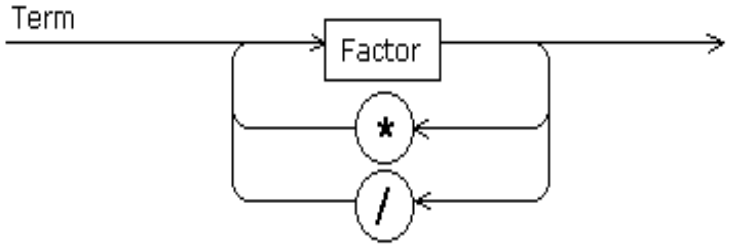
- Information in BNF and EBNF rules can be represented in the form of directed graph. Such graphs are called Syntax graph or syntax chart. It is also called railroad diagram.
- A separate graph is used for each syntactic unit (in the same way a variable in a grammar represents a separate unit)
- Rectangles represent non terminal symbols (syntactic unit)
- Circles contain terminal symbols
- Each rule is represented by a path from the input on the left (variable) to the output on the right.
- Any valid path from input to output represents a string generated by that rule.

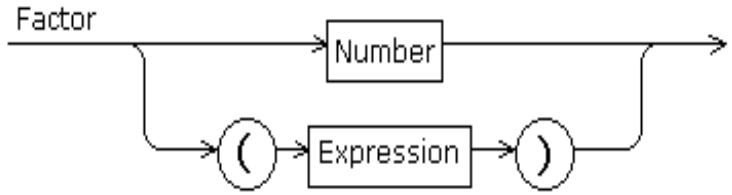
1. <assignment statement> ::=<variable>=<arithmetic expression>
2. <arithmetic expression> ::=<term>{[+|-]<term>}*
3. <term>::=<primary>{[/|X]<primary>}*
4. <primary>::=<variable>|<number>|(<arithmetic expression>)
5. <variable>::=<identifier>|<identifier>[<subscript list>]
6. <subscript list>::=<arithmetic expression>{,<arithmetic expression>}*

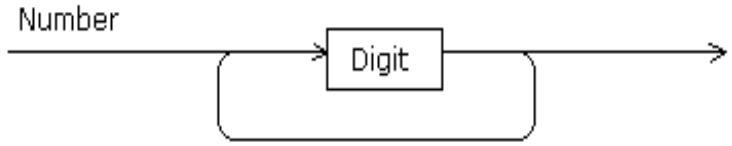


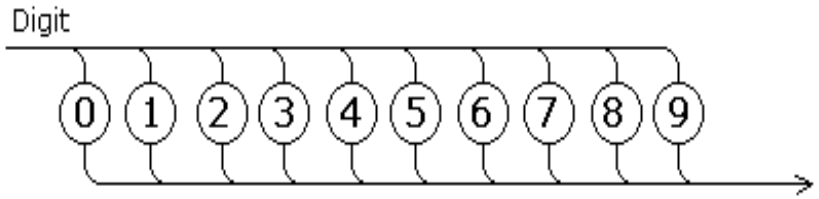
Convert to EBNF





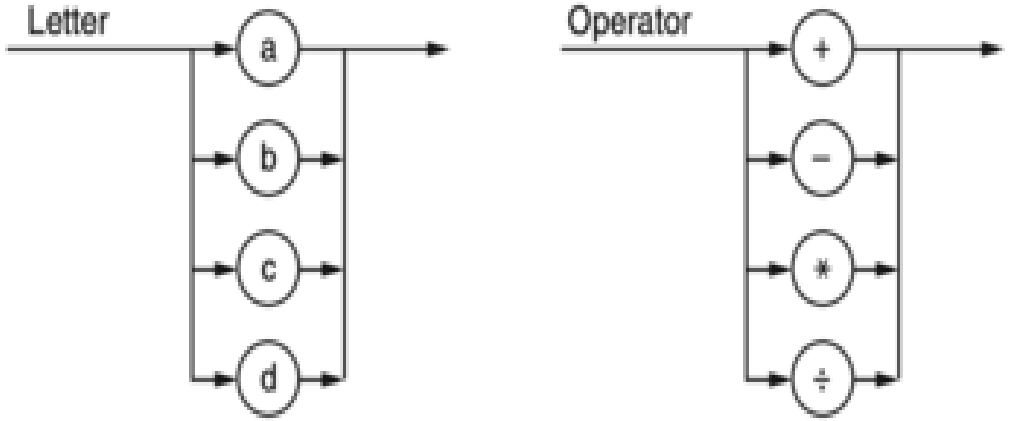







Assignment statement
Variable
=
Variable
Operator
Variable
;


Variable
Letter
Letter
Letter


Letter
a
b
c
d

Operator
+
−
*
÷

Derivations, Parse trees, Ambiguity

- Using a grammar, we can generate sentences. The process is called **derivation**

  Example: The simple grammar

  S → SS | (S) | ( )

  generates all sequences of paired parentheses.

  The rules of the grammar can be written separately:

  Rule1: S → SS

  Rule2: S → (S)

  Rule3: S → ( )

  One possible derivation is:

  S → (S) by Rule1

  → (S S) by Rule2

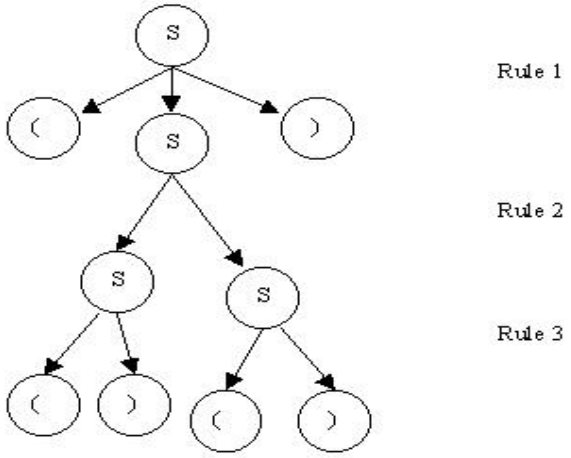
  → (( ) S) by Rule3

  →(( ) ( )) by Rule3

- The strings obtained at each step are called **sentential forms**. They may contain both terminal and non-terminal symbols. The last string obtained in the derivation contains only terminal symbols. It is called a **sentence** in the language.
- This derivation is performed in a ***leftmost*** manner. That is, at each step the leftmost variable in the sentential form is replaced.

PARSING

Parsing is the process of syntactic analysis of a string to determine whether the string is a correct sentence or not. It can be displayed in the form of a **parse tree***:*



•Each non-terminal symbol is expanded by applying a grammar rule that contains the symbol in its left-hand side.

• Its children are the symbols in the right-hand side of the rule.

•The order of applying the rules depends on the symbols to be expanded.
At each tree level we may apply several different rules corresponding to the nodes to be expanded.

Ambiguity

- The case when the grammar rules can generate two possible parse trees is called ambiguity for one and the same sequence of terminal symbols.

Example:

- Let **S** denote a statement, **Exp** - an expression.
  Consider the following rules for ***if*** statements (the words **if, then, else** are terminal symbols):
  Rule1: **If_statement** → if Exp then **S** else **S**
  Rule2: **If_statement** → if Exp then **S**

- Consider now the statement
  if a < b then if c < y then write(yes) else write(no);

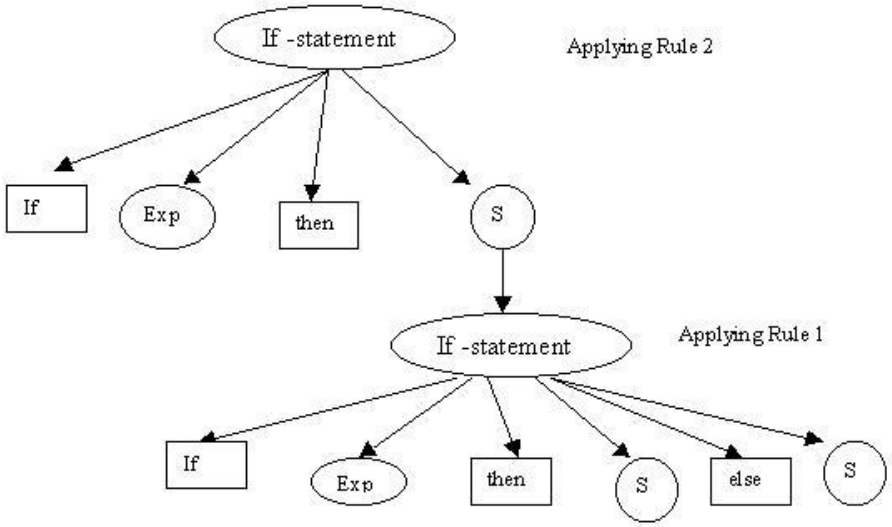
  Following the grammar rules above, there are two possible interpretations:

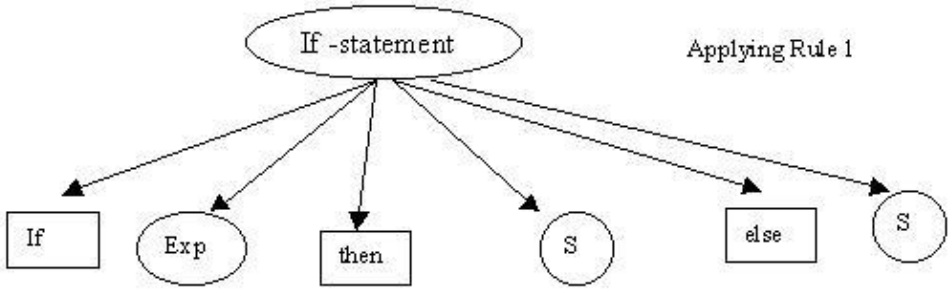
1. If a < b then

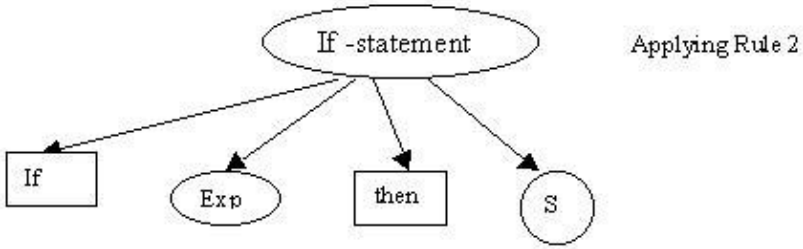
   if c < y then write(yes)
   else write(no);

2. If a < b then **if c < y then write(yes)**

   else write(no);







A grammar that contains such rules is called **ambiguous**.

It is very important that grammars for programming languages are not ambiguous.

Grammars for programming languages

There are 4 types of grammars depending on the rule format.

- **Regular grammars**: (Type 3)

  A → a

  A → aB

- **Context-free grammars** (Type 2)

  A → any string consisting of terminals and non-terminals

- **Context-sensitive grammars** (Type 2)

  String1 → String2

  String1 and String2 are any strings consisting of terminals and non-terminals, provided that the length of String1 is not greater than the length of String2

- **General grammars** (Type 0)

  String1 → String2, no restrictions.

Regular grammars and regular expressions

- **Regular expressions:** Strings of symbols may be composed of other strings by means of
  - **or –** choice of string/symbols , also denoted by U.
  - **concatenation** - appending two strings, and
  - **Kleen star operation** - any repetition of the string. e.g. a* can be a, or aa, or aaaaaaa, etc

  Given an alphabet **∑ , regular expressions** consist of string concatenations combined with the symbols U and *, possibly using '(' and ')'. There is one special symbol used to denote an empty expression: Ø

- **Formal definition**:
  - Ø, Epsilon and each member of **∑** is a regular expression.
  - If α and β are regular expressions, then (α β) is a regular expression.
  - If α and β are regular expressions, then α U β is a regular expression.
  - If α is a regular expression, then α* is a regular expression.
  - Nothing else is a regular expression.

- **Example:**
  Let **∑ =** {0,1}. Examples of regular expressions are**:**
  - 0,1, 010101, any combination of 0s and 1s
  - 0 U 1, (0 U 1)1*
  - (0 U 1)*01

- **Regular languages** are languages whose sentences are regular expressions**.**
  Regular grammars are used to describe identifiers in programming languages and arithmetic expressions.

- **Context-free grammars** generate **context-free languages**.
  They are used to describe programming languages.

- **FSAs**Languages are described by grammars.
  Given a grammar and a sentence, is the sentence in the language described by the grammar?
  In order to answer that question, we need an algorithm that takes as input
  - Grammar
  - sentence

  and gives a result:

  - yes – if the sentence is in the language described by the grammar
  - no – otherwise

- The simplest model of a computing device that can perform the above task is FSA -
  a language recognition device

- **A FSA is defined by :**
  - An alphabet,
  - A set of internal states, one is initial, one or more - final,
  - A transition function, that maps an input (a letter in the alphabet) and a state to a state.

- **Input:** a string of symbols, written on a tape.

- **Operation:** in a given state, FSA reads a symbol and enters some other state.
  A string is accepted (recognized) by the FSA, if when the last symbol is read,
  the FSA enters one of its final states.
- The set of all strings over an alphabet, that are accepted by the automaton,
  is **the language accepted by the automaton**.
- Thus each automaton describes some language, i.e. it corresponds to a grammar.

Types of automata

There are four basic types of automata, distinguished by the following characteristics:

- FSA, have no memory
- Pushdown automata, in addition to the tape, they use a stack to read from and write to
- Linear-bound automata, read and write on a tape of finite length in both directions
- Turing machine, read and write on an infinite tape in both directions

Computers are linear-bound automata

- Each type of automata corresponds to a grammar type - the type of the languages it can recognize:
- FSA - regular grammars
- Pushdown automata - context-free grammars
- Linear-bound automata - context sensitive grammars
- Turing machine - unrestricted grammars

Issues

- Deterministic machines: for each pair *state* - *input symbol* there is only one possible transition.
- **Decidability, computability:** A language is decidable if we can build a machine that can answer the question: "Does a given sentence belong to the language or not?"

  A language is semidecidable, if the machine can answer the above question only when the sentence does belong to the language. If the sentence does not belong to the language, the machine loops.

  A function is computable if we can build a machine that computes the function for all possible values in its domain.

- **The Halting problem**

  We can construct an example of a that cannot be solved. There are unsolvable problems.

Formal Model of Language Semantics

Introduction

- The need for language semantics:
  - for the programmer - to know how to use language constructs
  - for the implementer - to know how to implement the language

- **Informal** descriptions of language semantics - provided in verbal form in the language manuals. May be ambiguous.

- **Formal description of language semantics -** studied theoretically, no satisfactory models have been produced so far.

Semantic Models

- Grammatic Model
  - grammar rules are paired with semantic rules.
  - Resulting grammars are called **attribute grammars**
  - See fig.4.3. and the example below, p.129 in text book

- Operational / Imperative Model
  - Describe the meaning of the language constructs in terms of machine states, i.e. memory and register contents

- Applicative Model
  - Treat programs as functions.
    Two methods to build applicative models:
  - denotational semantics and functional semantics.

Denotational semantics

A technique for describing the meaning of programs in terms of mathematical functions on programs and program components. Programs are translated into functions about which properties can be proved using the standard mathematical theory of functions. Based on Lambda calculus.

Lambda calculus

- Lambda calculus is a formal mathematical system devised by Alonzo Church to investigate functions, function application and recursion. It has provided the basis for developing *functional programming languages*. Lambda calculus also provides the meta-language for formal definitions in *denotational semantics*. It has a good claim to be the prototype programming language.

- More about lambda calculus:
  http://www.csse.monash.edu.au/~lloyd/tildeFP/Lambda/Ch/

- Axiomatic Model
  - Describe the meaning as pre-conditions and post-conditions.
    Used in program verifications

- Specification Model
  - Describe the relationship among various functions that implement a program.
  - Example of a specification model:
    Algebraic data types - describe the meaning in terms of algebraic operations,
    e.g. *pop(push(S,x)) = S*

Formal models of syntax, grammars and parsing are well studied and widely used in programming language definition and implementation. Formal syntax definition can be used to construct parsers automatically from the definition.

Formal models of semantics of programming language are not so successful. A lot of research is going on towards building semantics-directed compilers that would translate programs into machine language using a formal specification of the semantics of the programming language.

Attribute grammars, that associate with each non-terminal in the grammar a set of attributes, are one of the earliest semantic models (Knuth, Donald E., Semantics of context-free languages, Mathematical Systems Theory) and they are still in use. Their most beneficial feature is that they can be used for efficient automatic translation. However, attribute grammars are not sufficiently powerful to represent the entire semantics of programming languages - they are too tightly coupled with parse trees.

A very powerful formal model, that can represent any computable function, is the denotational semantics (Scott, D.S. and Strachey, C., Towards a mathematical semantics for computer languages, in Proc. Symp. Computers and Automata , pages 19-46. Polytechnic Press, NY, 1971.) However its practical application is quite complex .

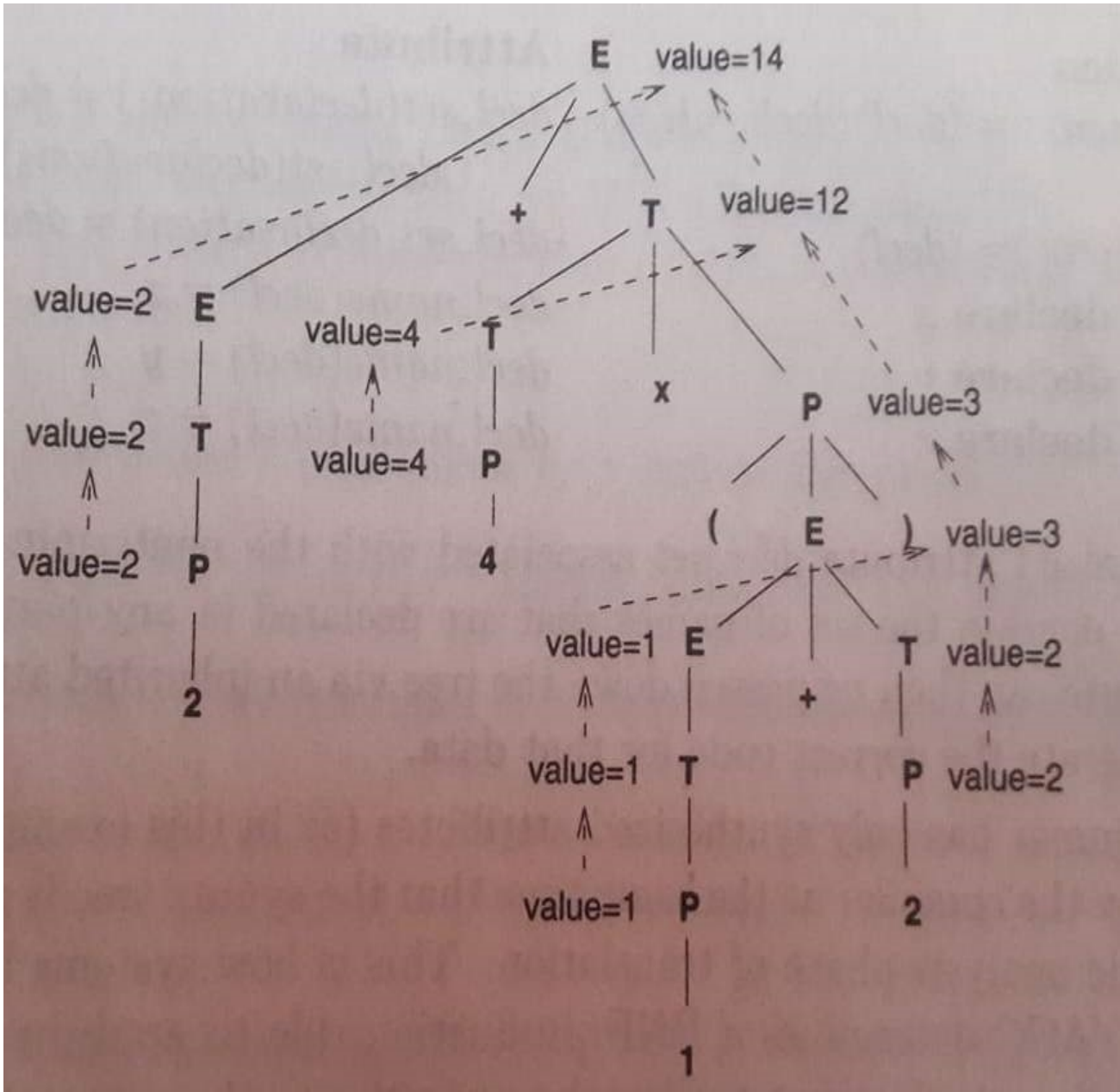
- Denotational Semantics
  - Lambda Calculus
  - Operation on Lambda Calculus-
  - Parameter Passing with Lambda Expression
  - Modelling Mathematics with Lambda Expression
    - Boolean Values
    - Integers

Attribute Grammar

- **Attribute grammar** is an extension of context-free **grammar**s in which each symbol has an associated set of **attribute**s that carry semantic information, and each production has a set of semantic rules associated with **attribute** computation.
- The idea is to associate a function with each node in the parse tree of a program giving the semantic content of that node.
- There are two types of attributes in attribute grammar.
  - Synthesized attribute
    - It is a function that relates the left hand side non terminal to values of the right hand side nonterminals. These attributes pass information up the tree.
  - Inherited attribute
    - It is a function that relates non terminal vales in a tree with non terminal values higher up in the tree. In other words, the functional value for the non terminals on the right of any rule are a function of the left hand side non terminal.
- Attribute grammar can be used to pass semantic information around the syntax tree.

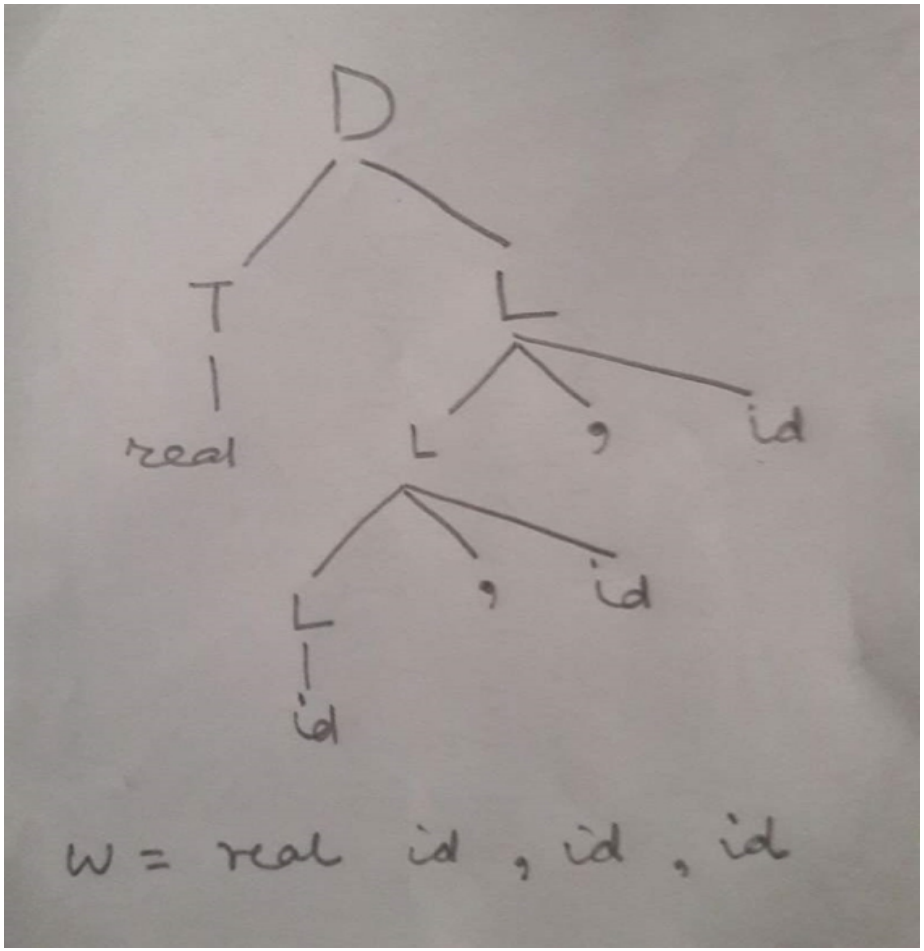
Production	Attribute
E→E+T	Val(E_1)=Val(E_2)+val(T)
E→T	Val(E)=val(T)
T→TxP	Val(T_1)=Val(T_2)+val(P)
T→P	Val(T)=val(P)
P→I	Val(P)=value of number I
P→(E)	Val(P)=val(E)

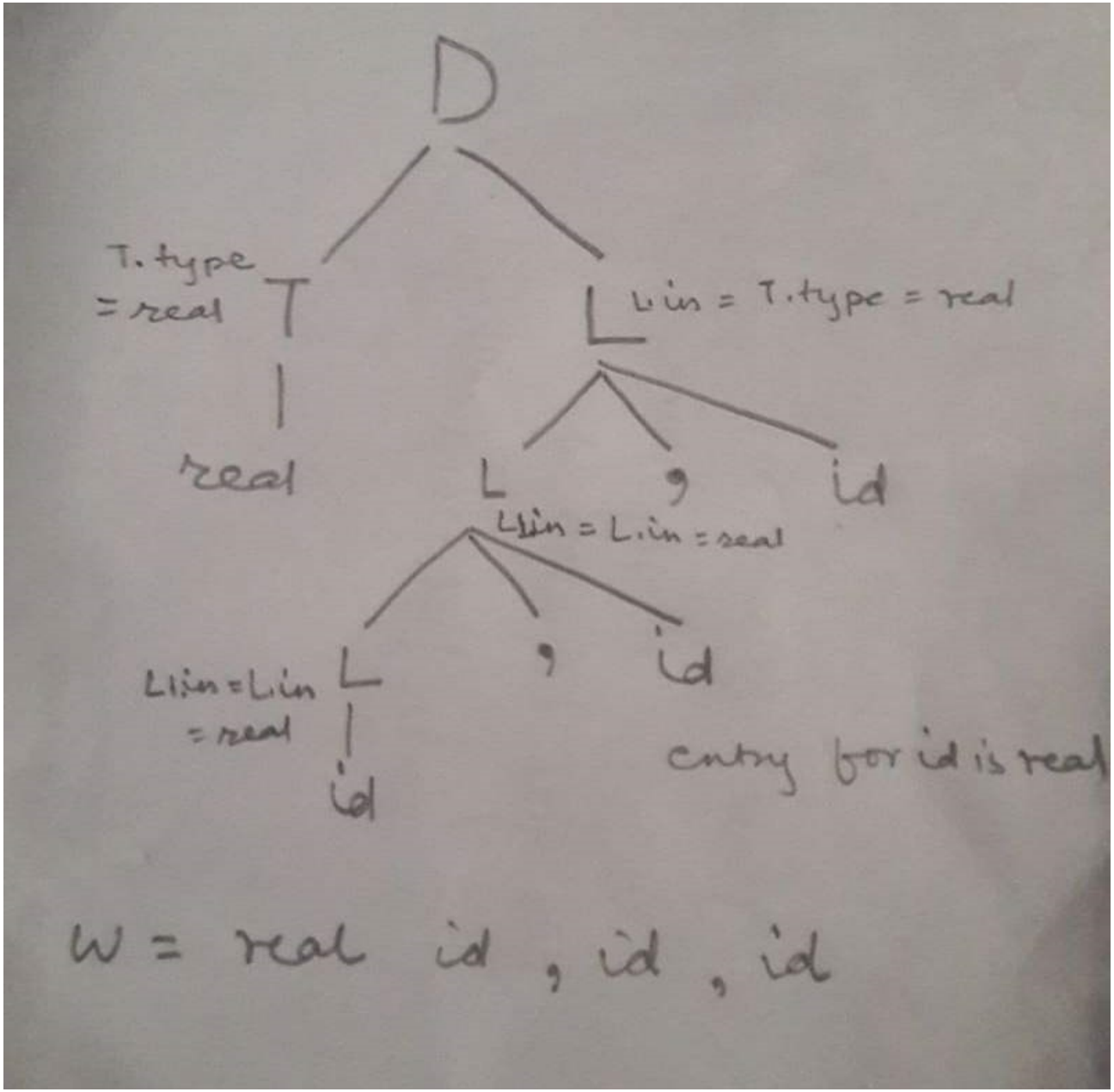
W= 2+4x(1+2)



D→TL	L.in = T. type
T→ int | real	T. type= int | real
L→L1,id	L1.in = L.in and addtype (id.entry, L.in)
L→ id	addtype (id.entry, L.in)

W= real id,id,id




D
T.type
= real
T
real
L
L.in = T.type = real
L
,
id
L1.in = L.in = real
L1.in = L.in
= real
L
id
,
id
entry for id is real
W = real id , id , id