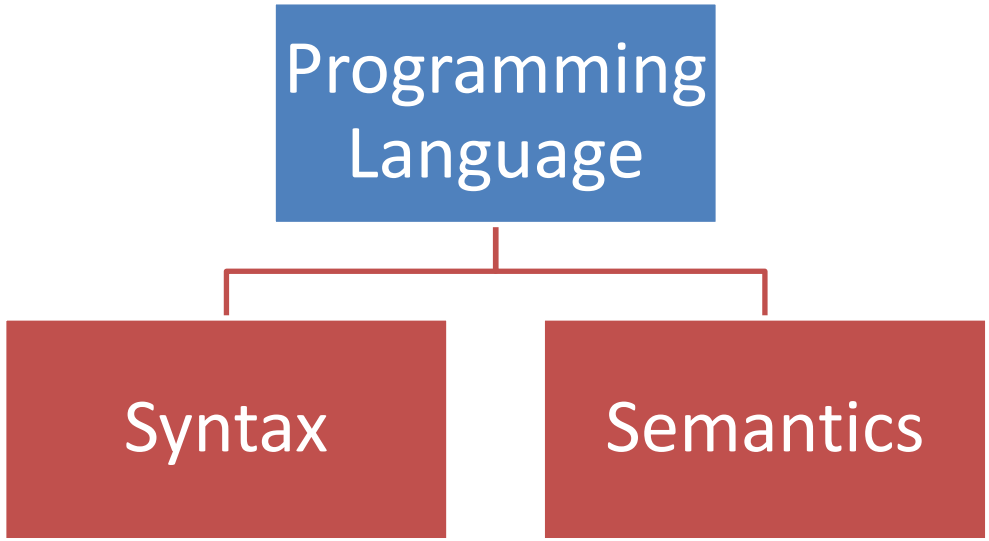
CHAPTER 2

LANGUAGE DESIGN PRINCIPLES

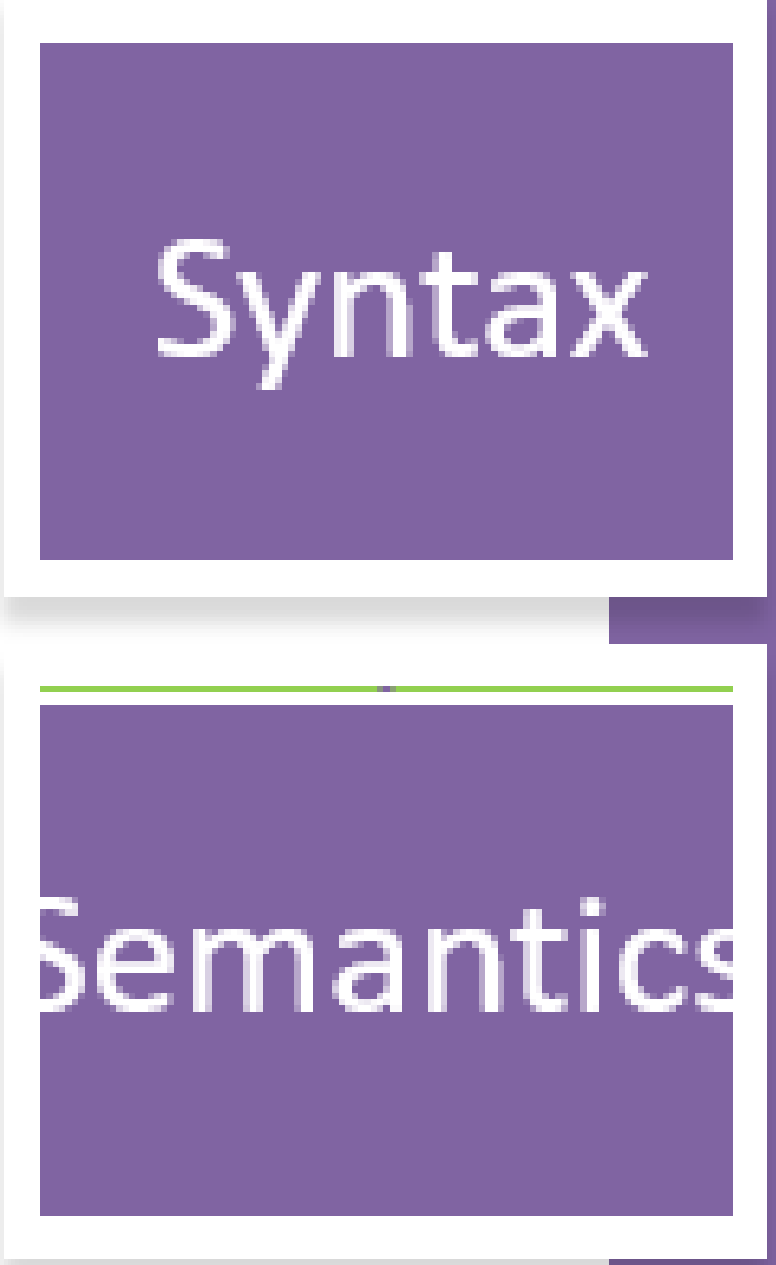
2.1 Introduction

- The study of **programming languages**, like the study of natural languages, can be divided into examinations of syntax and semantics.



2.1 Introduction

- The **syntax** of a programming language is the form of its expressions, statements, and program units.
    - Its **semantics** is the meaning of those expressions, statements, and program units.



2.1 Introduction

- For example, the **syntax** of Java **while** statement is

  `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. Otherwise, control continues after the **while** statement.

- **Syntax** and **semantics** are closely related. In a well-designed programming language, semantics should follow directly from syntax; that is, the form of a statement should strongly suggest what the statement is meant to accomplish.

- Describing **syntax** is easier than describing **semantics**.

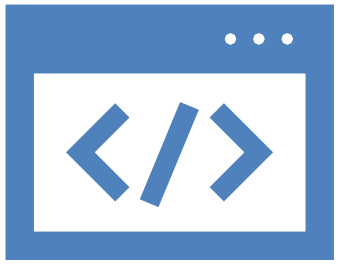
2.2 Describing Syntax

A **language**, whether natural (such as English) or artificial (such as Java), is a set of strings of characters from some alphabet. The syntax rules of a language specify which strings of characters from the language's alphabet are in the language.

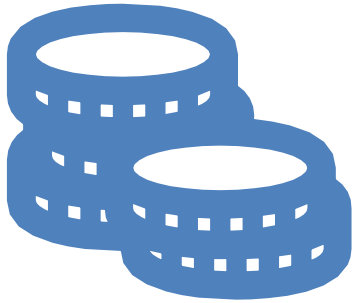
A **sentence (**or statement) is a string of characters.

A **language** is a set of sentences (or statements).

2.2 Describing Syntax



A **lexeme** is the lowest level syntactic unit of a language. The lexemes of a programming language include its numeric literals, operators, special words and others. One can think of programs as strings of lexemes rather than of characters.



A **token** of a language is a category of its lexemes.

2.2 Describing Syntax

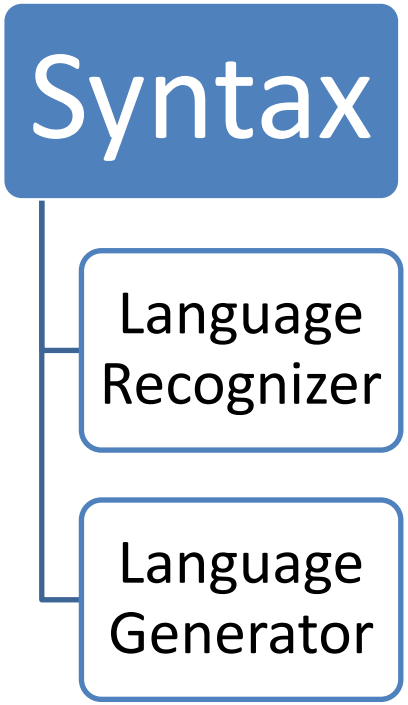
- Consider the following Java statement:

```
number = 2 * count + 17;
```

Lexemes	Tokens
number	identifier
=	equal-sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

2.2.1 Language Recognizers and Language Generators

- There are two formal approaches to describe **syntax**:



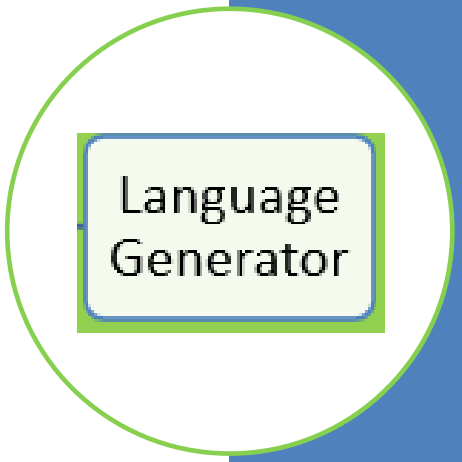
Given a string, a recognizer for a language L tells whether the string is in L.

A recognizer determines whether the given programs are in the language and syntactically correct.

Syntax analysis is a part of compiler. It is a recognizer for the language where the compiler translates.

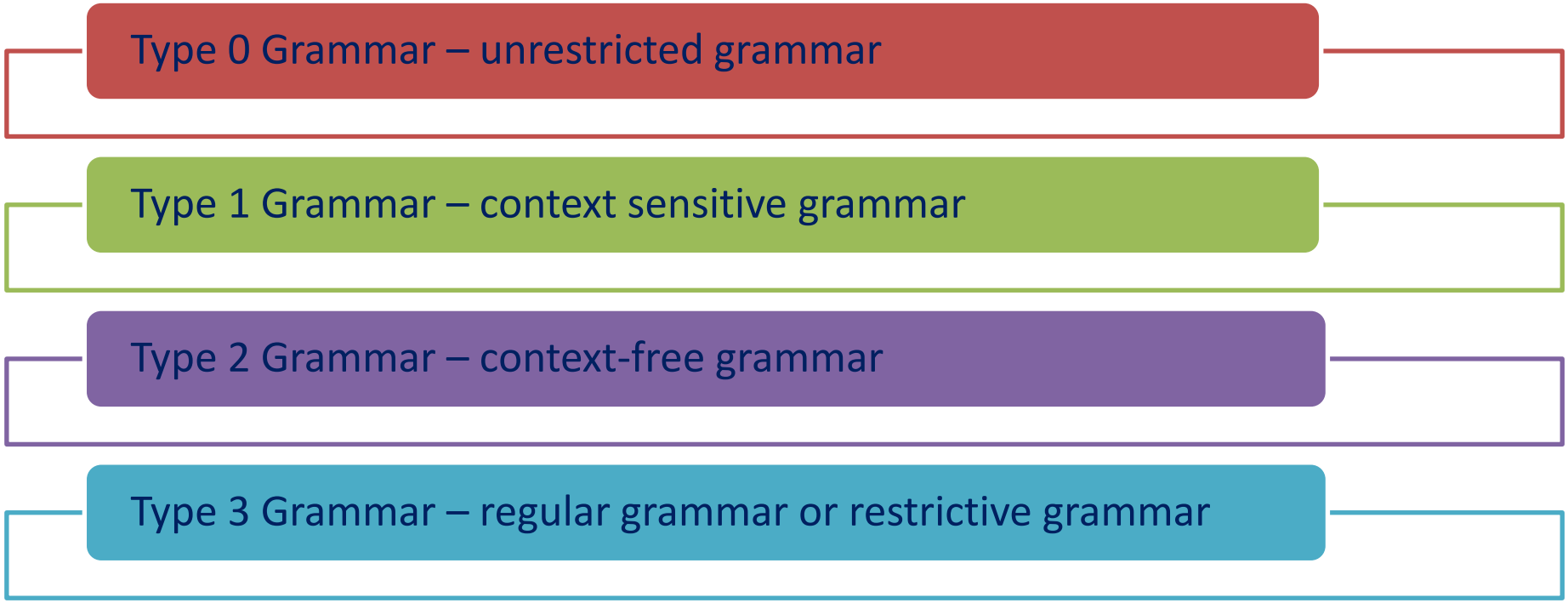
Syntax analyzer is also known as Parser.

A language generator can be used to generate the sentences of a language.



2.2.2 Classification of Grammars

- Four classification of Grammars:



- Grammars are commonly used to describe the syntax of programming languages.

2.2.3 BNF

The context-free grammar (Type 2 Grammar) is useful for describing the syntax of programming languages.

The revised method of context-free grammar is known as **Backus-Naur Form**, or simply **BNF** – introduced by Peter Naur and John Backus.

BNF is a popular method for describing programming language syntax.

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

A **BNF** description, or grammar, is simply a collection of rules.

Although **BNF** is simple, it is sufficiently powerful to describe nearly all of the syntax of programming languages.

BNF uses abstractions for syntactic structures.

A simple Java assignment statement, might be represented by the abstraction <assign>. The actual definition of <assign> can be given by

<assign> -> <var> = <expression>

- An Ada **if** statement can be described with the rule

```
• <if_stmt> -> if <logic_expr> then <stmt>
• <if_stmt> -> if <logic_expr> then <stmt>
  else <stmt>
```

or with the rule

```
• <if_stmt> -> if <logic_expr> then
  <stmt>| if <logic_expr> then <stmt>
  else <stmt>
```

2.2.4 Grammars and Derivation

- A grammar is a generative device for defining languages.
- The sentences of the language are generated through a sequence of applications of the rules.
- A sentence generation is called a derivation.

- Example 2.1 : A Grammar for a Small Language

```
<program> -> begin <stmt_list> end
<stmt_list> -> <stmt>
             | <stmt>; <stmt_list>
<stmt> -> <var> = <expression>
<var> -> A | B | C
<expression> -> <var> + <var>
          | <var> - <var>
     | <var>
```

- This language has only one statement form – assignment. A Program consists of the special word **`begin`**, followed by a list of statements separated by semicolons, followed by the special word **`end.`**

- An expression is either a single variable, or two variables separated by either a + or – operator. The only variable names in this language are A, B, and C.

- Example 2.2 – A Grammar for Simple Assignment Statements

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

- The grammar of example 2.2 describes assignment statements whose right sides are arithmetic expressions with multiplication and addition operators and parentheses.

For example, the statement

```
A = B * ( A + C )
```

is generated by the leftmost derivation:

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

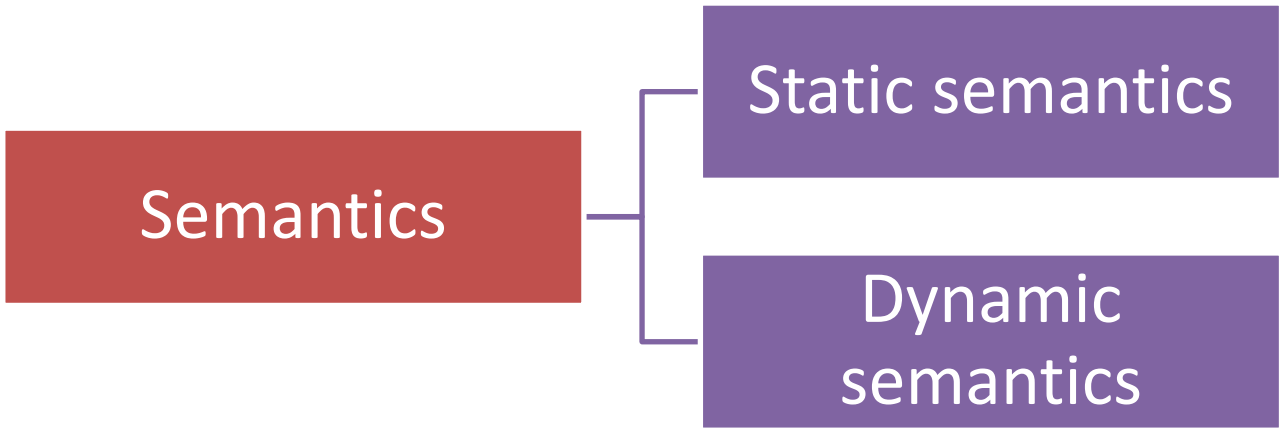
2.2.5 Parse Tree

- Parse trees can be used to describe the hierarchical syntactic structure of the sentences of the languages.
- A Parse Tree for the assignment statement

```
A = B * ( A + C)
```

2.3 Describing Semantics

- Generally, semantics reveals the meaning of the syntax or grammar. Semantics can be categorized into two types:



Static semantics

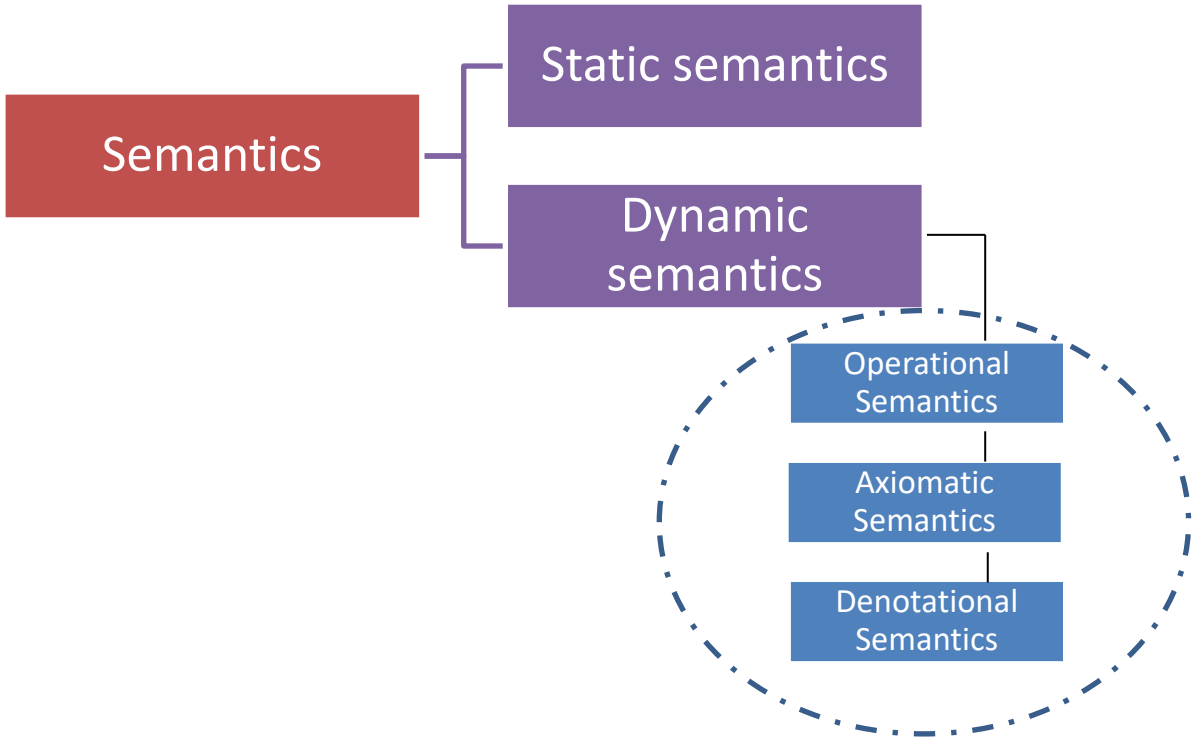
- The **static semantics** of a language is only indirectly related to the meaning of programs during executions; rather, it has to do with the legal forms of programs (syntax rather than semantics). The analysis required to check the specifications can be done at compile time.

Dynamic semantics

- **Dynamic semantics** can be done at run time.

- The reason why one might be concerned with describing semantics is that programmers obviously need to know precisely what statements of a language do.

- Three formal methods of describing **dynamic semantics** - operational semantics, axiomatic semantics, and denotational semantics.



Operational Semantics

Describe the **meaning of a program** by executing its statement on a machine, either real or simulated. The changes that occur in the machine's state when it executes a given statement define the meaning of that statement.

Axiomatic Semantics

- Defines a programming language behavior by **applying mathematical logic** and the development of method to prove the correctness of programs.

- Axiom semantics is based on mathematical logic. The logical expressions are called predicates, or assertions.

Denotational Semantics

- Defines a programming language behavior by **applying mathematical functions** to programs and program components to represent their meaning.

- It is based on recursive function theory.

2.4 Identifiers/ Names

- An identifier or a name is a string of characters used to identify some entity in a program. Identifiers or names are associated with variables, labels, subprograms, parameters, and others. The term **identifier** is often used interchangeably with **name**. Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and underscore character (_).

- C++ : letters, digits, underscore
- Pascal : letters, digits, underscore
- Java : letters, digits, underscore

- **Variables**

A variable is a name or an identifier which is associated with a memory location to store a value. The value can be changed.

- **Reserved Words**

Reserved words (or keywords) are special words in the programming language that cannot be used as names (or identifiers). The reserved words cannot be redefined in any program; that is, they cannot be used for anything other than their intended use.

- **Constants**

A constant stores only one value. Its value cannot be changed by assignment or by an input statement during program execution. Using a constant to store fixed data, rather than using the data value itself, has one major advantage.

When the fixed data changes, you do not need to edit the entire program and change the old value to the new value. You just need to change at one place (declaration statement).

- Examples of declarations:

- C++ : `const int MAX = 100;`

- Pascal :
```
const
  MAX = 100;
```

- Java : `final int MAX = 100;`

2.5 Variables

- A program variable is an abstraction of a computer memory cell or collection of cells. Programmers often think of variables as names for memory locations, but there is much more to a variable than just a name.
- The move from machine languages to assembly languages was largely one of replacing absolute numeric memory addresses with names, making programs far more readable and therefore easier to write and maintain.

- The **address of a variable** is the **memory address** with which it is associated.

- The **type of a variable** determines the **range of values** the variable can have and the set of operations that are defined for values of the type.

- The **value of a variable** is the **contents of the memory cell** or cells associated with the variable.

- The **scope of a variable** is the range of statements in which the **variable is visible**. A variable is visible in a statement if it can be referenced in that statement.

- The **lifetime of a variable** refers to the **period of time that the variable is bound to an address.**

- The binding of a variable to a value at the time it is bound to storage is called **initialization.**

2.6 Data Types

2.6.1 Primitive Data Types

- Data types that are not defined in terms of other types are called **primitive data types**. Nearly all programming languages provide a set of primitive data types.

A. Numeric Types

Many early programming languages had only numeric primitive types.

Integer

The most common primitive numeric data types is integer.

C++ : `int, long`

Pascal : `Integer, Longint`

Java : `int, long`

Floating-Point

Floating-point data types model real numbers, but the representations are only approximations for most real values.

C++ : `float, double`

Pascal : `Real`

Java : `float, double`

Decimal

Most larger computers that are designed to support business systems applications have hardware support for decimal data types.

Decimal data types store a fixed number of decimal digits, with the decimal point at a fixed position in the value. These are the primary data types for business data processing and therefore essential to COBOL.

B. Boolean Types

Boolean types are perhaps the simplest of all types. Their range of values has only two elements, one for true and one for false. Boolean types are often used to represent switches or flags in programs. Although other types, such as integers, can be used for these purposes, the use of Boolean types is more readable. C++, Java and Pascal have Boolean data types.

C++ : `bool X;`

Pascal : `X : Boolean;`

Java : `boolean X;`

C. Character Types

Character data are stored in computers as numeric codings. Traditionally, the most commonly used coding was the 8-bit code ASCII (American Standard Code for Information Interchange), which uses the values 0 to 127 to code 128 different characters. ISO 8859-1 is another 8-bit character code, but it allows 256 different characters. A 16-bit character set named Unicode includes the characters from most of the world's natural languages.

C++ : `char code;`

Pascal : `code : char;`

Java : `char code;`

2.6.2 Character String Types

- A character string type is one in which the values consist of sequences of characters. Character strings are also an essential type for all programs that do character manipulation.

- If strings are not defined as primitive type, string data is usually stored in arrays of single characters. This is the approach taken by C and C++. In Pascal, String data type is provided.

C++ : `char name[30];`

Pascal : `name : String;`

Java : `String name;`

2.6.3 User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers. In some languages, users can define two kinds of ordinal types : enumeration and subrange.

A. Enumeration Types

An enumeration type is one in which all of the possible values, which are named constants, are provided in the definition.

C++ :

```
enum  sizes { small, medium,
large, jumbo };
sizes drink_size, popcorn_size;
drink_size = large;
popcorn_size = jumbo;
```

B. Subrange Types

A subrange type is a contiguous subsequence of an ordinal type. Subrange types were introduced by Pascal and are included in Ada.

Pascal :

```
type
Letter = 'A' .. 'Z';
DaysInMonth = 1 .. 31;
```

2.6.4 Array Types

- An array is a group of contiguous memory locations that all have the same name and the same type.
- Each element is identified by its position in the array, relative to the first element. A reference to an array element in a program is called an array subscript.

C++ :

```
int num [10];
float salary [12];
```

Pascal :

```
num : Array[1..10] of Integer;
      salary : Array[1..12] of Real;
```

Java :

```
int data[ ] = new int[10];
float salary[ ] = new float[12];
```

2.6.5 Record Types

- A record is a collection of related items. Unlike an array, however, the individual components of a record can contain data of different types. The record data types are used for file/ data base applications.

- Records have been part of all of the most popular programming languages, since the early 1960's, when they were introduced by COBOL. In C and C++, records are supported with the **struct** data type. In Pascal, we can use **record** data type.

Cobol

Cobol :

```
EMPLOYEE-RECORD.
EMPLOYEE-NAME.
05 FIRST                PICTURE IS
   X(20).
05 MIDDLE      PICTURE IS X(10).
05 LAST        PICTURE IS X(20).
       02      HOURLY-RATE   PICTURE
   IS 99V99.
```

C++

C++ :

```
struct Student
{
   int id;
   char name [30];
   float cgpa;
};
Student stu1;
Student array1 [20];
```

Accessing members of a structure (struct) :

```
   cin >> stu1.id;
   cout << stu1.id << endl;
   cin >> array1[m].cgpa;
   cout << array1[m].cgpa << endl;
```

Pascal

Pascal :

```
type
Student = record
Id : Integer;
Name : String;
cgpa : Real;
                end;
var
                stu1 : Student;
                array1 : array [1..20]
   of Student;
```

Accessing members of a record:

```
                ReadLn (stu1.Name);
                WriteLn ('Name = ',
   stu1.Name);
                ReadLn
   (array1[m].Name);
WriteLn ('Name = ', array1[m].Name);
```

2.6.6 Pointer Types

- Pointers contain memory addresses as their values. Pointers have been designed for two distinct kinds of uses. First, pointers provide some of the power of indirect addressing, which is heavily used in assembly language programming. Second, pointers provide a method of dynamic storage management.

C++ :

```
int *iptr;
float *yptr;
circle *circleptr;
```

2.7 Expressions and Assignment Statements

Expressions are the fundamental means of specifying computations in a programming language.

In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls.

The purpose of an arithmetic expression is to specify an arithmetic computation.

- The operators can be **unary**, meaning they have a single operand, or **binary**, meaning they have two operands.
- Examples (C++) :
- Unary operator : `x++` `x--` `++x` `--x`
- binary operator : `a + b` `a * b` `a > b` `a < b`

Operator

Operator	C++	Pascal
Assignment operator	=	:=
Relational operators	< > <= >= == !=	< > <= >= = <>
Arithmetic operators	+ - / * %	+ - / * mod div
Logical operators	&& ||	and or

An assignment statement can simply cause a value to be copied from one memory cell to another. But in many cases, assignment statements include expressions with operators.

Examples (C++) :

```
x = 10;
y = (x + b) / (k * p);
```

A relational operator is an operator that compares the values of its two operands.

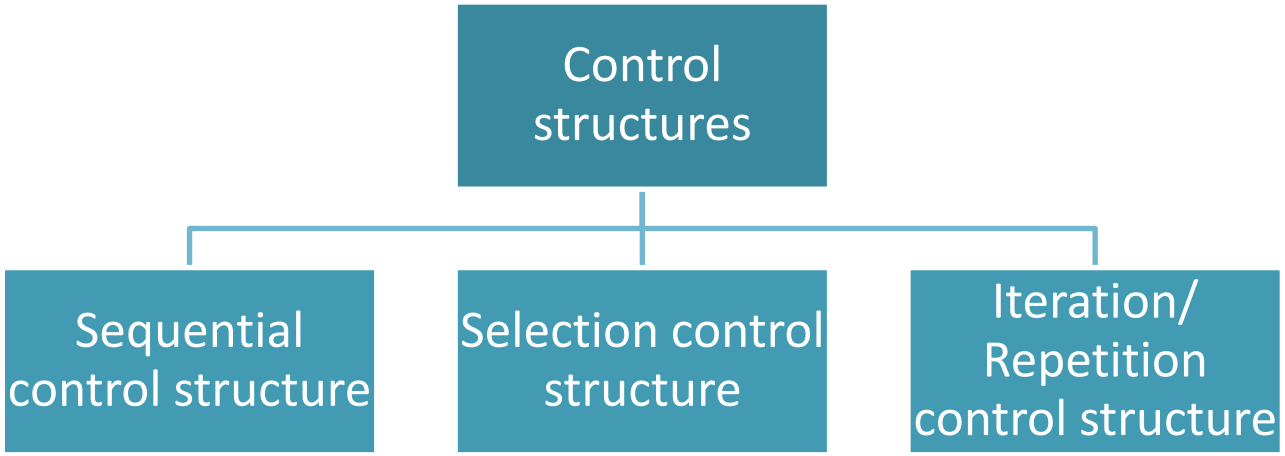
A relational expression has two operands and one relational operator.

Examples :`a > b` `a < b` `a >= b` `a <= b`

- A **short-circuit evaluation** of an expression is one in which the result is determined without evaluating all of the operands and/or operators. For example, the value of arithmetic expression (13 * a) * (b / 13 – 1) is independent of the value (b / 13 – 1) if a is 0, because 0 * x = 0 for any x. So, when a is 0, there is no need to evaluate (b / 13 – 1).

2.8 Control Structures

- There are three types of control structures in the programs:



2.8.1 Sequential Control Structure

- The sequential control structure is the simplest of all the structures. The program statements are executed sequentially – one by one, starting from the first statement until the last statement.

- The basic operations in a program:

- input → process → output

2.8.2 Selection Control Structure

In selection control structures, the program executes particular statements depending on some condition(s). Certain statements are to be executed only if certain conditions are true.

There are different types of selection statements such as one-way selection, two-way selection, and multiple selection.

2.8.2 Selection Control Structure

Examples of statements:

C++ :if statement, switch statement

Pascal :if statement, case statement

Java :if statement, switch statement

2.8.3 Iteration/ Repetition/ Looping Control Structure

- An **iterative** statement is one that causes a statement or collection of statements to be executed zero, one, or more times. Every programming language has included some method of repeating the execution of segments of code.

2.8.3 Iteration/ Repetition/ Looping Control Structure

Examples of statements:

C++: for statement, while statement, do while statement.

Pascal: for statement, while statement, repeat statement

Java: for statement, while statement, do while statement

There are different types of loops that can be implemented in programs:

Counter-controlled loops

The loop is controlled by a control variable and the number of repetition is known.

Sentinel-controlled loops

The loop continues to execute as long as the program has not read the sentinel value.

Flag-controlled loops

A flag-controlled loop uses a Boolean variable to control the loop.


Thank
you!!!