ch.1

Benefits of studying concepts of programming language:

(1) increasing the ability to describe programs
(2) increasing the ability to choose appropriate programming language
(3) increasing the ability to learn new language
(4) understanding the significance of implementation
(5) improving the design of programming language

programming domains:

(1) Scientific applications — large numbers
(2) Business applications — decimal arithmetic and character data
(3) artifical intelligence — symbolic computations
(4) Systems programming — computer devices and external devices
(5) web software — markup language

Evaluating programming languges: (evaluating criteria)

(1) Readability (2) Writability (3) Reliability (4) Maintainability
(5) Efficiency (6) Portability (7) Generality

- Simplicity (simple): if it consists of a small number of basic constructs to be learned. (readability ↑)
- Expressivity (expressive): if it consists of a set of constructs that perform a lot of computations (writability ↑)
- Orthogonality: if a relatively small set of primitive constructs can be combined in a relatively small number of ways to define the language behaviour. orthogonality = simplicity + expressivity
- control statements enhances readability and writability
- The presence of good ways to define datatypes and structures in a language improves readability
- restriction on identifier length — readability ↓
- using end loop and end if instead of } — readability ↑
- using special words as variable name — readability ↓
- Static / const — readability ↓
- abstraction — writability ↑
  ↳ is the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored. → (reliability ↑)
- Type checking is testing for type errors in a given program, either by the compiler or during program exec.
- Exception handling is the ability of a program to intercept runtime error, take corrective actions and then continue according to the program specifications. (reliability ↑)

- Aliasing is having two or more distinct names in a program that can be used to access the same memory cell.
  ↳ excessive usage of aliasing may reduce reliability & readability
- Compiler takes a program written in the source language and translates it into a program written in machine language (the executable). [increase the runtime efficiency of the program] (reliability ↑)
- Interpreter takes a program written in the source language and executes its statements line-by-line
- Interpreted programs are much slower than compiled programs.

programming languages categories:

(1) Imperative / procedual languages — data and procedures (c)
(2) object-oriented languages — processing with data objects and control access to data (c++)
(3) Scripting languages — interpreted not compiled (python)
(4) Declarative / functional / logic languages — Rule based languages in no particular order (prolog)
(5) Markup languages — languages used to describe something (XHTML)
(6) Special-purpose languages — languages dedicated for specific applications

ch.2

- Syntax: is the form of programming languages' expressions, statements and programming units.
- Semantic: is the meaning of programming languages' expressions, statement and programming units.
- Lexems: sequence of small units in a statement.
- token: category of the lexems.
- Grammars are formal mechanisms used to describe the syntax of programming languages
- nonterminal <a> is an abstract symbol that helps describe part of the grammar but cannot appear in any Program
- Derivation: generating a specific program (leftmost derivation)
- parse tree: hierarchical structure of a program
- Ambiguity: if there exists a program (or part of program) that has two or more distinct parse tree compatible with a given grammar.
- operator precedence is the order of associating operands to operations in the same statement.
- precedence determines which operator gets its operands first.
- Operator with higher precedence should appear lower in the parse tree.
- operator associativity: is the order of associating operands to operators having the same precedence
  ↳ for operators with equal precedence, associativity determines which operator gets its operands first
- Left associative are associated to their close operands in their order from left to right.
- Right associative are associated to their close operands in their order from right to left.
- Extended BNF is a similar grammar to BNF which attempts to improve its readability and writability
- {.} n≥0 [.] 1 or more ( ) must choose [ ] optional ↳ no associativity
- attribute grammars are extensions to BNF grammars that can describe some aspects of syntax.
  ↳ attributes associated with some grammar symbols
  ↳ attribute computation function associated with some grammar rules to specify how attribute values are computed
  ↳ predicates associated with some grammar rules to specify additional syntatic rules
- intrinsic attributes associated with leaf nodes and get their values determined from outside the parse tree
- lookup function lookup a variable name in the symbol table and returns the variable's type.
- operational semantic is a method to describe the semantics of a program construct
  ↳ readability ↓ writability ↑

ch.3

- special words are used to make programs more readable
  (1) keywords that can be redefined to be used in other purposes, which reduces readability
  (2) reserved words that cannot be used for any other purposes, such as variable names
- A name is a string of characters used to identify some entity in a program.
  ↳ reserved word cannot be used as a name ↳ has no length limit to increase readability
- Two common naming styles are my_small_stack and the camel notation mySmallStack.
- Some programming languages force naming rules to improve readability
  ↳ variables in PHP must begin with $ ↳ variables in Perl starts with $ or @ or % (→ int, → array, → hash table)
- case sensitive: uppercase and lowercase letters in names are considered distinct. (readability ↓)
- A program variable is an abstraction of a computer memory cell (→ readability ↑)
  ↳ variables were used as names for memory locations replacing absolute numeric memory addresses
  (1) name (may be accessed by knowing its address
  (2) type (type of a variable determines its size)
  (3) address: is the machine memory address with which it is associated (l-value) → left hand side of =
  (4) value: is the contents of the memory cell associated with the variable (r-value) → variable / const value
  (5) scope: range of statements where the variable is visible in source program (spatial)
  (6) lifetime: is the time durations at execution which the variable physically exists in memory (temporal)
- Constant value has no address
- Aliases: variables having the same type and address
- named constants is a variable that is assigned a constant value only once before runtime and remains unchanged during every execution of the program (readability ↑)
- readonly variable is assigned a value in its constructor and does not change until its destruction
- named constants differs from read only variable (C++ does not have both, readability ↓)
- binding is an association between an entity and an attribute
- binding time is the time at which binding takes place
  (1) compile time: binding occurs during compilations (static) (unchanged throughout program execution)
  (2) load time: binding occurs when the program is loaded into memory and ready to run (static)
  (3) run time: binding occurs while the program is running (dynamic) (can change)
- Type binding:
  (1) Static type binding: variables may bind statically before runtime
    ↳ explicit declaration: a statement in a program that explicitly declares a variable and its type
    ↳ implicit declaration: type of a variable is implicitly deduced from the first appearance of the variable name

(2) Dynamic type binding: a variable binds to its type only when an assignment statement having the variables as its LHS is executed during run time. (writability ↑ reliability ↓)

- Dynamic type binding is most costly / inefficient than static type binding
- Allocation: the process of binding a variable to its memory cell
- deallocation: the process of unbinding a variable from its memory cell] the in-between is lifetime
- Storage binding:

(1) Static variables: load time → execution termination (global variables & name constants)
↳ can be accessed by direct addressing / no allocation or deallocation / does not support recursive subprogra.

(2) Stack-dynamic variables: run time → called function terminates its execution
↳ Elaboration of a declaration statement refers to the storage allocation and binding process indicated by the decl.

(3) Explicit heap-dynamic (disorganized) variables: nameless variables are allocated by explicit run-time instructions.
↳ bind statically to their types, but they bind dynamically to storage (new / delete)
↳ C++ must be explicit, in Java are implicitly deallocated by the Java run-time garbage collector

(4) implicit heap-dynamic variables: bind to heap storage only when they are assigned values

- value binding: at run time except when a static variable is initialized (load time)
- Scope rules: determines how a particular occurrence of a name is associated with a variable
- Referencing environment of a statement is the collection of all variables that are visible in that statement
- local variable: if it is declared there
- non local variable: visible within the program unit or block but are not declared there (global variables)
- Static scoping: the scope of a variable can be statically determined prior to execution.
- Dynamic scoping is based on the calling sequence of subprograms at run time
↳ readability ↓ reliability ↓ efficiency ↓
- Typechecking: is the activity of ensuring that they are of compatible types.
- Compatible type: to be implicitly converted by the compiler or interpreter to a legal type
- Coercion: automatic conversion • type error: if there is an incompatible type
- Static type checking is better than dynamic type checking (reliability ↑)
- Type equivalence: if they are considered compatible without coercion
↳ name type equivalence: if they are defined in declarations that use the name type
↳ structure type equivalence: if their types have identical structure
- Side effect: occurs when the function changes either one of its parameters or a global variable
- Short-circuit evaluation: the result is determined without evaluating all parts of the expression