Programming Methodology

Spring 2009

Introduction

Topics

Course information

TAs, course web site, grading, exams, assignments

Objectives of this course and class schedule

Introduction to programming

Qualities of software: depending on programming skill

Software/algorithm specification for programming

Programming languages

General information

- Instructor: 백윤흥 (ypaek@snu.ac.kr)
  - Office hour: by prior appointment thru e-mail
- Head TA: 양승준 (prog-ta@optimizer.snu.ac.kr, ☎: 880-1742)
- References
  - Lecture notes: main reference source
  - Textbooks for C/C++ programming
    - Deitel, *How to Program C++*, 6^{th} *ed.*, Prentice Hall
  - Textbooks for programming language concepts
    - R. Sebesta, *Concepts of Programming Languages*, Addison-Wesely
    - R. Sethi, *Programming Languages: Concepts & Constructs*, Addison-Wesley
- Class home page:

http://eng.snu.ac.kr/lecture

Lecture information

- Organization of the lecture
  - One regular lecture
  - One programming lab
- Regular lectures
  - Th
  - 10:30-12:10
- Programming lab hours
  - Fr
  - 13:00-15:00

Tentative grading policy

- Two exams: 60 %
- Assignments: 30%
  - For every programming hour, one pair of programming assignments will be handed out.
  - One is *in-class* and the other is *at-home*.
  - mostly simple programming assignments using the C++ language
  - Details will be given by the TAs later.
- Class attendance: 10%
  - Attendance sheets will be handed out during the regular lecture and the programming lab hour.
  - Please sign it up for your attendance verification.

    → But, do not do it for your friend(s)!!

Objectives of this course

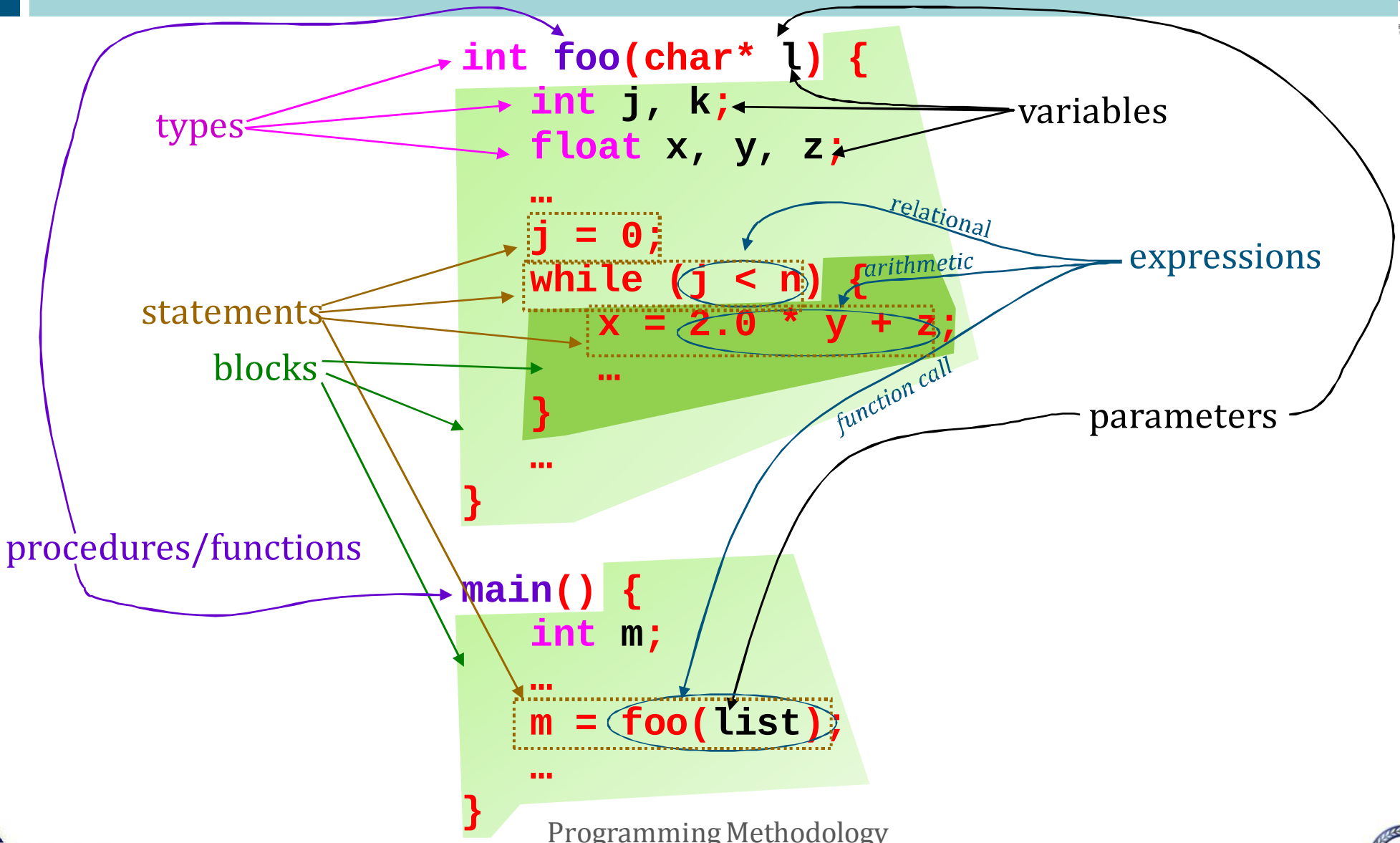
- make it better to understand languages you have been using
- allow better choice of programming languages as an engineer who needs programming to solve his/her problem
- increase vocabulary of programming constructs
- write better programs
- practice various programming constructs of C/C++, the most popular languages ever used in engineering societies
- ultimately, help you fulfill the course requirements and get a better job.

  → *Every engineer today surely needs a good programming skill!*

What to cover

- Principles of programming for software development
- Basic programming language concepts and constructs
  - types, polymorphism, coercion, overloading
  - expressions, assignments, conditional statements
  - procedures/functions, parameter passing
- Moving towards Object Oriented Programming
  - blocks, storage managements scope, binding
  - modules, data abstraction, object abstraction
- Exercising OOP with C++, subsuming C
  - getting used to diverse programming constructs common in most existing languages
  - practice object-oriented programming as well as imperative one

Programming language constructs



Tentative class schedule

week	lecture	*Programming lab*
1	Class information & programming basics	*C-related features in C++*
2		
3	Types, Polymorphism	
4	Types, Polymorphism	*C++ basics*
5	Variables, Scopes, Binding, Parameters	
6	Variables, Scopes, Binding, Parameters	
7	Control Structures (expressions, statements, functions)	
8	Blocks, Modules, Data abstraction	Exam 1
9	Blocks, Modules, Data abstraction	*OOP practice with C++*
10	Object abstraction, Object-oriented programming	
11	Object abstraction, Object-oriented programming	
12	Object abstraction, Object-oriented programming	
13	Additional issues on OOP with C++	
14	Additional issues on OOP with C++	*Advanced OOP practice*
15	Exam 2	

Topics

Course information

TAs, course web site, grading, exams, assignments

Objectives of this course and class schedule

Introduction to programming

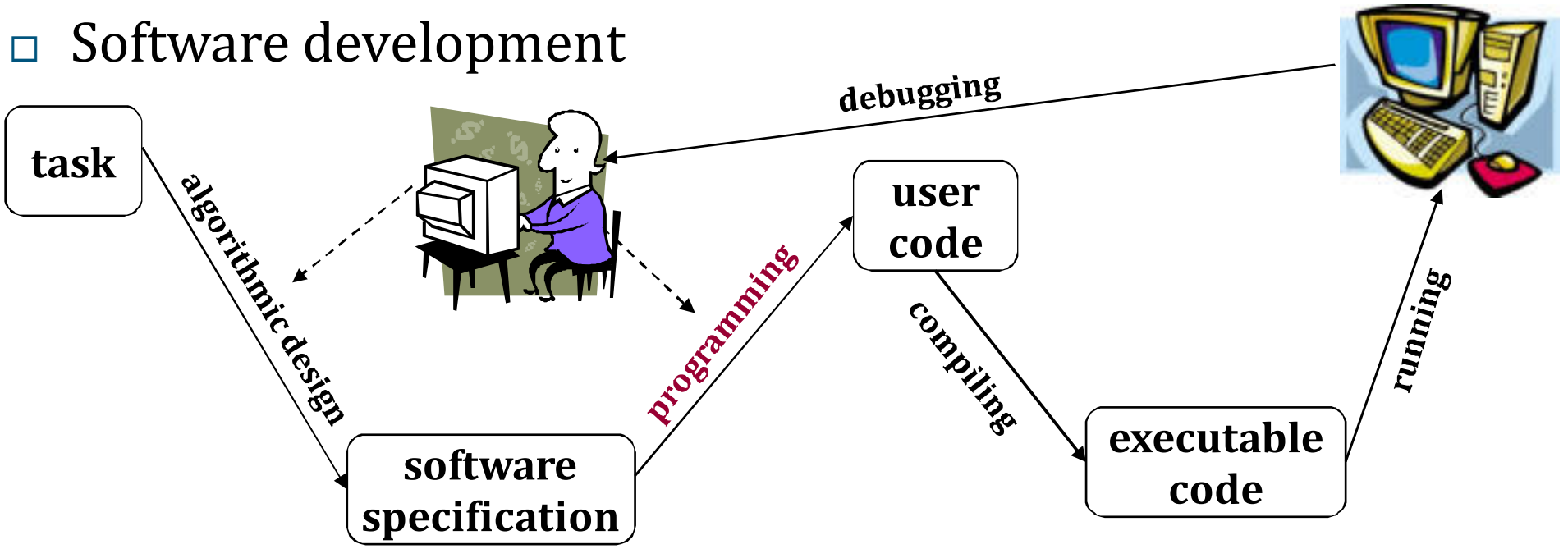
Qualities of software: depending on programming skill

Software/algorithm specification for programming

Programming languages

What is programming?

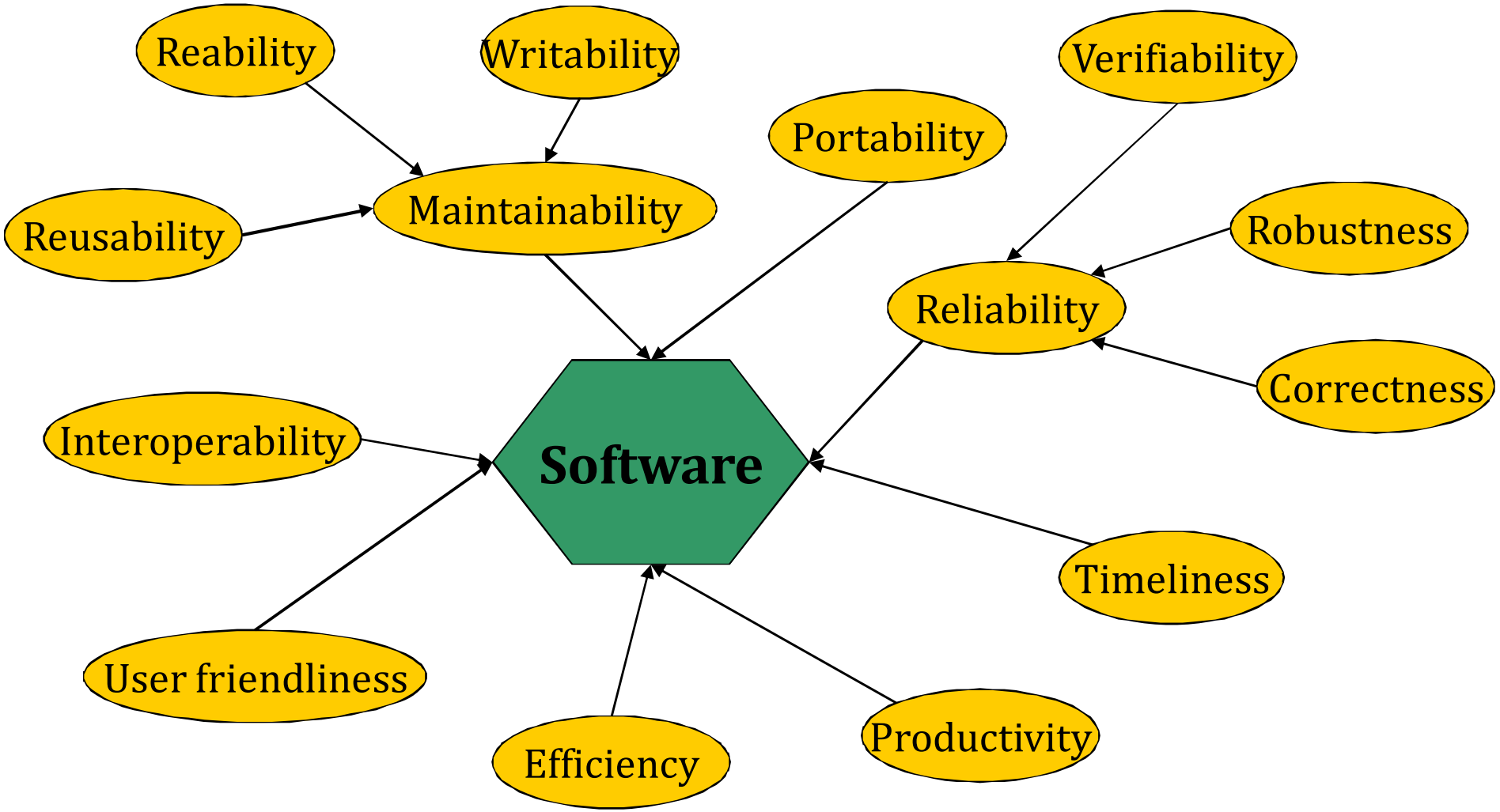
- Software development



- Programming…
  - is an essential part of software development.
  - converts an algorithmic description of a user task into the code that can be executed on the machine.

    → *Qualities of software* are determined by how well the code is written.

Representative qualities of software



Reliability

- An application must perform its functions *correctly* as expected.
  - Unfortunately, there is no formal way to *verify* a software product is absolutely correct. → Most products are not absolutely correct
  - Release 1 of a product is usually buggy, and software products are thus commonly released along with a list of 'Known Bugs'.
  - But, they are considered to be reliable if a software error is minor.
- A program is said to be *robust* if it behaves reasonably even in unanticipated (mostly, erroneous) circumstances.
  - hard to measure because even a correct program may not be robust, and not all unexpected situations can be tested for the program.
  - A good program should be well prepared for ill-formatted input.

```
scanf("%d",b);
if (a / b < 0)
   …
```

Are this code is robust?
What if **b** *is zero?*
How to make this code more robust?

Efficiency (≈ performance)

- Efficient use of time/resource for computation
  → typical measures: latency, throughput, memory/disk space
- Meaning of efficiency is changing.
  - Memory was once scarce and expensive decades ago.
  - CPUs today are several orders of magnitude faster.
  - Some systems such as embedded systems impose strict constraints on memory space and CPU performance.
- Ways to evaluate or predict the performance of a system
  - Analysis of the complexity of algorithms
    Ex) find a key from a sorted list **`key_list`**

```
for (i = 0; i < n; i++)
  if (key_list[i] == key)
    return i; /* The key is found */
```

Time complexity = O(n)

```
j=0;  k=n;   i=n/2;
while (i > 0) {
  if (key_list[i] = key)
    return i;
  if (key_list[i] > key) k = i;
  else j = i;
  i = (k - j) / 2;
}
```

Time complexity = O(*log* n)

n/2,n/4,…,4,2,1

  - Simulation

User friendliness

- User friendly interface
  - important in some systems
  - window/mouse-based GUI
    → MS Windows vs. MS DOS, X-windows for Unix
  - less critical in embedded and scientific applications
- User friendly programming environment
  - express algorithms with more intuitive high-level constructs
    → **`for`**, **`while`**, **`if`**, **`<`**,**`>`**,**`*`** (C/C++ vs. machine/assembly code)
  - write a program in the natural mathematical sense
    → functional vs. imperative languages
  - simplify programming by describing only what (not how) to do
    → logic vs. imperative
  - application domain specific supports
    → Ex) Java/C# with full of APIs & libraries for web/Windows apps

Portability

- Capability of software to run in different environments
- Compatibility issue
  - Some code is not executable on different machines
    → assembly vs. high-level languages
  - Some code makes assumptions on machine facilities.
- Performance portability issue
  - Machine-dependent assumptions make the code less performance-portable.
- Solutions?
  - smart compilers, flags for each machine platform, …

designed for 32-bit machines

```
int l, n, m;
l = 1024;  /* = 2^10 */
m = 4096;  /* = 2^12 */
n = l * m; /* = 2^22 */
```

Even if this is high-level language code, it is still incompatible to 16-bit machines

Ex: *Old programs were usually programmed with many complex data **compaction** or **overlay** schemes to save memory space because memory was once expensive. Such programs are not efficient these days when virtual memory is supported, and memory is abound and cheap.*

Maintainability

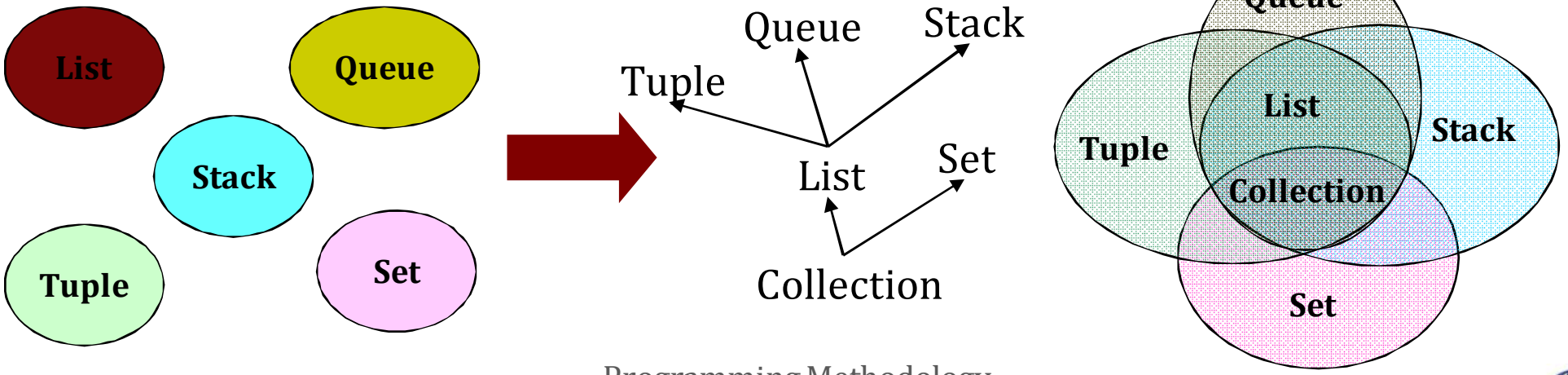
- Software maintenance…
  - means the modifications (read/write) that are made to a software system after its initial release.
  - is a very important factor of software design since…

  total costs of software = 40% of initial development + 60% of maintenance.
- Two properties of maintainability
  - Repairability (20% of the software maintenance cost)
    - removing residual errors → *corrective maintenance*
    - easier to repair a program with well-designed modules than a monolithic program
  - Evolvability (80% of the maintenance cost)
    - adjusting the application to changes in the environment
      → *adaptive maintenance*
    - changing the software to improve some of its qualities
      → *perfective maintenance*

Reusability

- part of maintainability, akin to evolvability
- use an existing product to build another new product
  - UNIX shell: existing commands easily extensible with new user programs
  - Libraries: the same library routines called by other codes after linking

    ➔ ocx, drv, dll: libraries in MS Windows
- Object-oriented design: base class reused for derived class
  - Ex) Code for several types of collection



Software specification

- Every software system must be carefully specified before it begins to be actually programmed.
  → A clearly, well-defined specification will ease programming and reduce errors in the final software product.
- Software specification is important since the software design based on a well-defined specification can achieve better qualities of software than a naïve, bruteforce design.
  → *ensuring correctness and robustness, reducing product development time, increasing maintainability and reducibility, …*
- Specification styles
  - Descriptive → The desired properties (not behavior) are stated.
  - Operational → Software is specified by describing the desired *behavior*, which is described by providing an abstract model of the software that in some way can simulate its behavior.

Software specification styles

- Descriptive/declarative specification
  - quite formal → easier to verify the specification
  - Example: the sorting

    → An array **b[0:n]** is the result of sorting an array **a[0:n]** if **b** is a permutation of **a** such that for all **i**, $0 \le i < n$, $b[i] \le b[i+1]$.
  - less flexible → limited applicability to various applications
- Operational/procedural specification
  - rather informal, more flexible to describe software
  - Example: the sorting of an array **a**
    1. Let **a[0:n]** be an array to be sorted.
    2. Allocate another array **b[0:n]** which will store the sorted result of **a**.
    3. Find the minimum of **a** and remove it from **a** to place it at **b[0]**.
    4. Find the minimum of the array of the remaining **n** elements of **a** after removing its minimum and remove it from **a** to place it at **b[1]**.
    5. Repeat Step 4 with **b[2]**… until all **n** elements of **a** have been removed.

Descriptive specifications

- Operational specifications describe *how* the software system is designed to work.
- Describing 'how' is low level way to specify software thus…
  - flexible but informal
  - usually more difficult to verify the specification
  - maybe more time consuming and error-prone
- Descriptive specifications define what the software system should perform by stating its desired properties.
  - applicable when the detailed 'how' is not needed to achieve the task.
  - most effective when 'how' is not a decisive factor for the performance.
- Methods for descriptive specification
  - Logic specifications
  - Algebraic specifications

Logic specifications

- based on a mathematical formula of a First-Order Theory (or predicate calculus) that is …
  → *an expression involving variables, numeric constants, functions and predicates, all of which are connected via logical operators* ($\wedge$, $\vee$, $\neg$, $\rightarrow$, $\equiv$)
- Examples
  - $x / 1 = x$
  - $x > y \wedge y > z \rightarrow x > z$
  - $x > 5 \vee x < 1$
  - $x = y \equiv y = x$
  - $\forall x, y, z\ (x > y \wedge y > z \rightarrow x > z)$
  - $\forall x\ (\exists y\ (y = x + z))$
- types of variables
  - bound variables: quantified in the formula
  - free variables: not bound variables

Specifying a program with FOT

- Mathematical formulas are used to express program properties in the following form:

  {input: i_1, i_2, …, i_n}
  F(*input args* i_1,…,i_n; *output args* o_1,…,o_m)
  {output: o_1, o_2, …, o_m, i_1, i_2, …, i_n}

  - F is a function in the program;
  - i_j's are input argument variables and o_k's output ones;
  - {input/output: v_1,v_2,…,v_x} is a set of formulas involving v_i's.
- Example

  Design a program **div_multiple** *which, given two integers x and y, produces the division x / y if x is a multiple of y.*

Logic specification does not say anything about how the code is implemented.

{$\exists n\ (n,i_1,i_2 \in Z \wedge i_1 = i_2 * n)$}
div_multiple(in int i_1,i_2; out int o_1)
{$o_1 \in Z \wedge o_1 = i_1 / i_2$}

implementation

specification

```
int div_multiple(int i1, i2) {
    int r = i1 / i2 * i2;
    if (r == i1)
        return i1 / i2;
    /* else is undefined */
}
```

More examples

- A function that produces the greater of two integers

  $\{true\}$
  greater(in int i_1,i_2; out int o_1)
  $\{(o_1 = i_1 \vee o_1 = i_2) \wedge o_1 \geq i_1 \wedge o_1 \geq i_2\}$

- What does this program **func** do?

  (assume that all variables are integers)

  $\{i_1 > 0 \wedge i_2 > 0\}$
  func(in int i_1, i_2; out int o_1)
  $\{(\exists\, z, y\ (i_1 = o_1 * z \wedge i_2 = o_1 * y)) \wedge \neg\ (\exists\, h\ (\exists\, v, w\ (i_1 = h * v \wedge i_2 = h * w) \wedge h > o_1))\}$

- A function that reverses the original sequence of an array?

Pros and cons of logic specifications

- Quite formal specification, thus verification of a given specification is facilitated.

  Ex) Prove this specification for **div_multiple1** is incorrect

  $\{\exists n\ (n,i_1,i_2 \in Z \wedge i_1 = i_2 * n)\}$

  div_multiple1(in int i_1,i_2; out int o_1)

  $\{o_1 \in Z \wedge o_1 = i_2 / i_1\}$

  $(\exists n\ (n,i_1,i_2 \in Z \wedge i_1 = i_2 * n)) \wedge (o_1 \in Z \wedge o_1 = i_2 / i_1)$

  $\equiv i_1,i_2 \in Z \wedge \exists n\ (n \in Z \wedge \mathbf{i_1 = i_2 * n}) \wedge (o_1 \in Z \wedge \mathbf{i_1 = i_2 * o_1^{-1}})$

  From the two expressions in red, we have $\mathbf{n = o_1^{-1}}$, *which implies* $\mathbf{o_1^{-1} \in Z}$.

  Conclusion: $\mathbf{o_1 \in Z \wedge o_1^{-1} \in Z}$ *implies that* **div_multiple1** *works correctly only when* $\mathbf{i_1 = i_2}$.
  So, we conclude "This specification is different from what we originally intended to design".

- less flexible so that only certain limited types of software systems can be described.
  - possibly quite complicated to describe long sequential operations
  - awkward to describe non-numerical programs such as text processing and information retrieval

Algebraic specifications

a generation of arithmetic

- Algebra, instead of logic, is used as the underlying mathematical formalism to specify the software system.
  - Properties of software are described by a collection of data sets together with operations on the sets.
  - Example: (data sets, operations)
    - (integers, {add, multiply, subtract, divide, …})
    - (strings, {new, add, length, concatenation, compare, …})
- Example of a syntax of the algebra

```
algebra handle_strings;
 sets Str, Char, Int, Bool;
operations
 new: ∅ → Str;
 add: Char x Str → Str
 concat: Str x Str → Str
 length: Str → Int
 compare: Str x Str → Bool
 reverse: Str → Str
end
```

Execution samples

new = “”
add(‘e’,new) = “e” = e|“”
add(‘e’,add(‘e’,new))) = “ee” = e|“e” = e|e|“”
length(add(‘e’,add(‘e’,new))) = 2
concat(“ee”,s|“nu”) = “eesnu”
compare(“ee”,add(e,add(e,new)) = true
reverse(“eesnu”) = “unsee”

Look similar to Class in C++?

Axioms: specification of operations

- Operations are described in a list of equations/functions, called *axioms*, that define the properties of the arithmetic (not logic!) operations on its data sets.
- Example of axioms for the program **handle_strings**

axioms for **handle_strings** with [Str s, t; Char c]

```
new = "";
add(c, s) = c | s;
concat(new, s) = s;
concat(add(c, s), t) = add(c, concat(s, t));
length(new) = 0;
length(add(c, s)) = 1 + length(s);
compare(new, new) = true;
compare(new, add(c, s)) = false;
compare(add(c, s), new) = false;
compare(add(c, s), add(c, t)) = compare(s, t);
```

Pros and cons of algebraic approach

- can be verified, but less formally than logic specification.
  - length("eesnu") = length(add('e',"esnu") = 1 + length("esnu") = 1 + length(add('e',"snu") = 2 + length("snu") = … = 5 + length(new) = 5
  - concat("ee","snu") = concat(add('e',"e"),"snu") = add('e',concat("e","snu") = add('e',add('e',concat(new,"snu")) = add('e',add('e',"snu")) = add('e',e|"snu")) = e|e|"snu" = "eesnu"
  - compare("eesnu","eesnu") = compare(add('e',"esnu"),add('e',"esnu")) = compare("esnu","esnu") = … = compare(new,new) = true
  - compare("eesnu","eesun") = compare(add('e',"esnu"),add('e',"esun")) = … = compare("nu","un") = compare(add('n',"u"),add('u',"n")) = ??

  → *The correctness is proved by less formal, case-by-case examinations!*

  → *add and new are not provable by reasoning. So they are assumed to be correct based on intuition.*

  → *The original axioms are incomplete since we cannot deduce the desired correctness of 'compare' for* **handle_strings***!*
- relatively more straightforward to write a program from an algebraic specification than a logic specification.

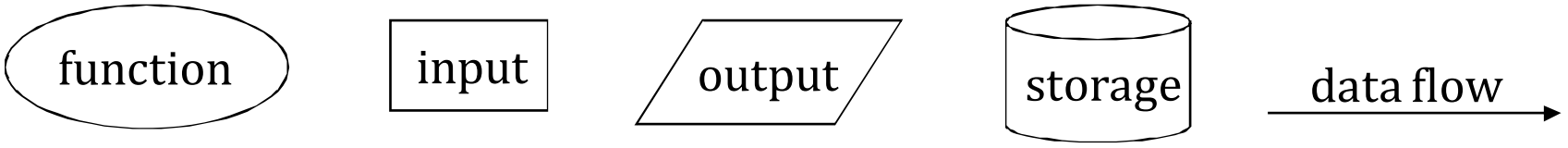
Operational specifications

- Notations for operational specification
  - Data flow diagrams
  - Finite state machines
  - Petri nets (➔ This won't be discussed in this lecture)
- Mostly notations are pictorial since graphical specifications can be more intuitive and easier to grasp than textual ones.

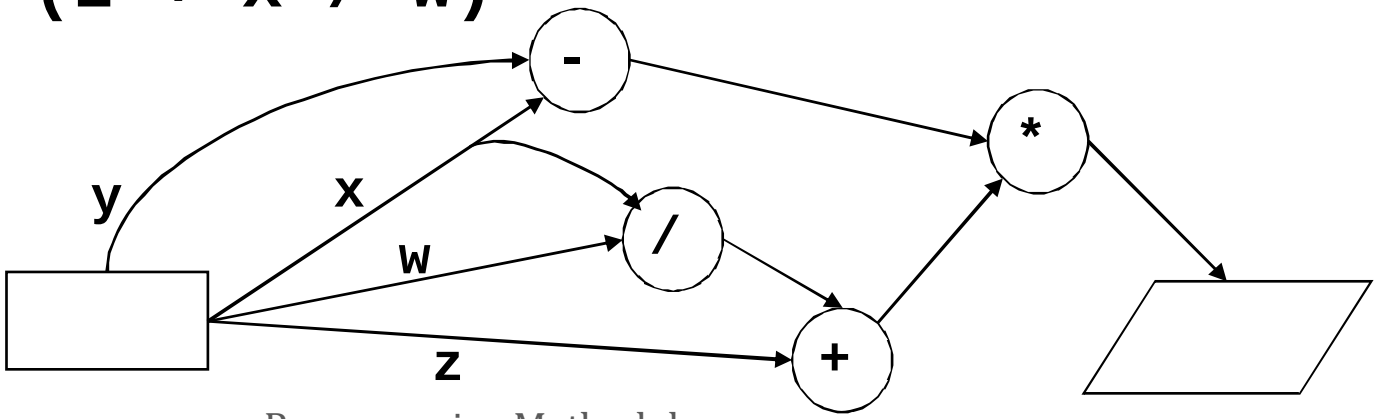
  → *A picture is worth a thousand words, according to folk wisdom!*
- Not a single notation works for specifying every S/W system.
  - It is important to understand advantages and disadvantages of each notation and to decide the most appropriate notation for the intended software system.
  - Often several notations are combined to specify the software.

DFD (data flow diagram)

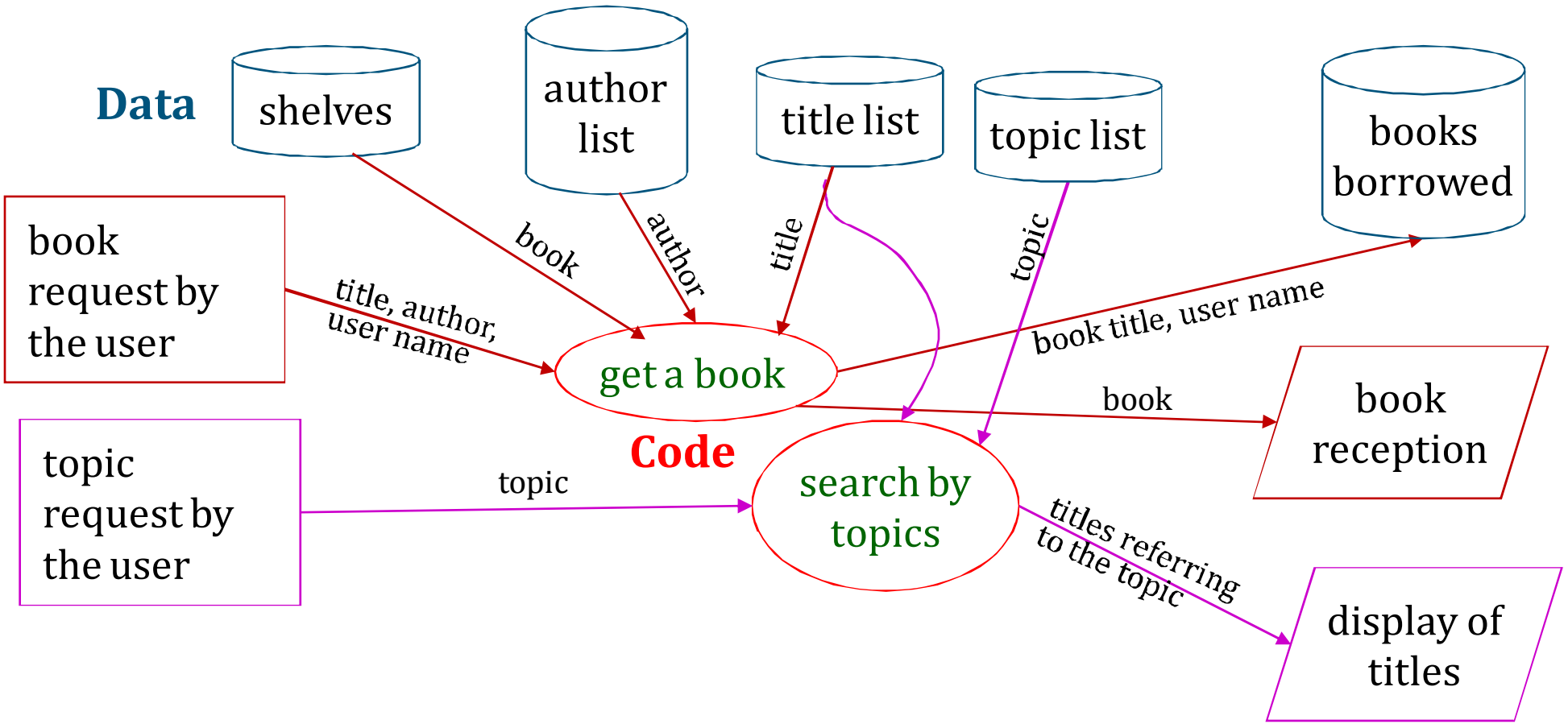
- is a widely used notation for specifying the functions of an *information system* (where various data are flowing), such as
  - data base systems
  - web-based information retrieval systems.
- 5 basic components



- Ex: `(x - y) *(z + x / w)`



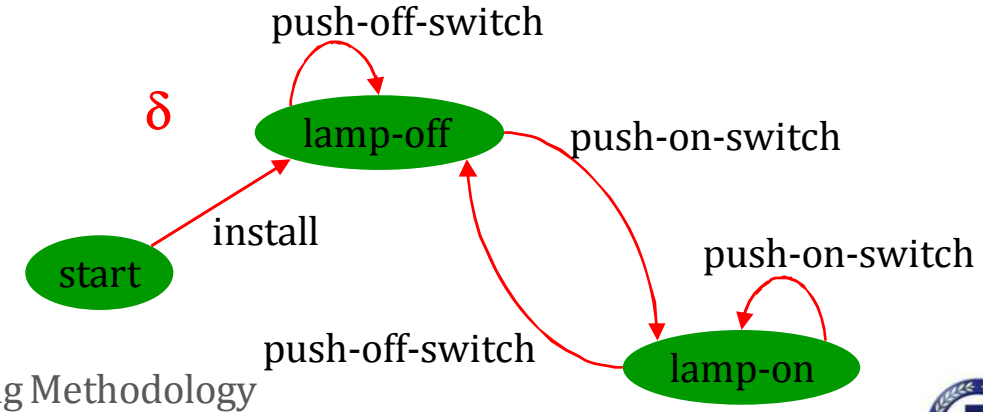
A library information system in DFD



FSM (finite state machine)

- In most software systems, *control flow* as well as *data flow* must be specified.
  - DFDs lack the capabilities to handle control flow information, such as external inputs or interrupts that alter or control the normal course of data flow.
  - FSMs are a simple, well-known model for describing control aspects.
- Basic components of a FSM
  - a finite set of states Q which tells the current status of the system
  - a finite set of inputs Σ which is input from outside or some action
  - a transition function, $\delta : Q \times \Sigma \rightarrow Q$, which tells how the system should react to the current input
  - Ex: a FSM for a lamp switch

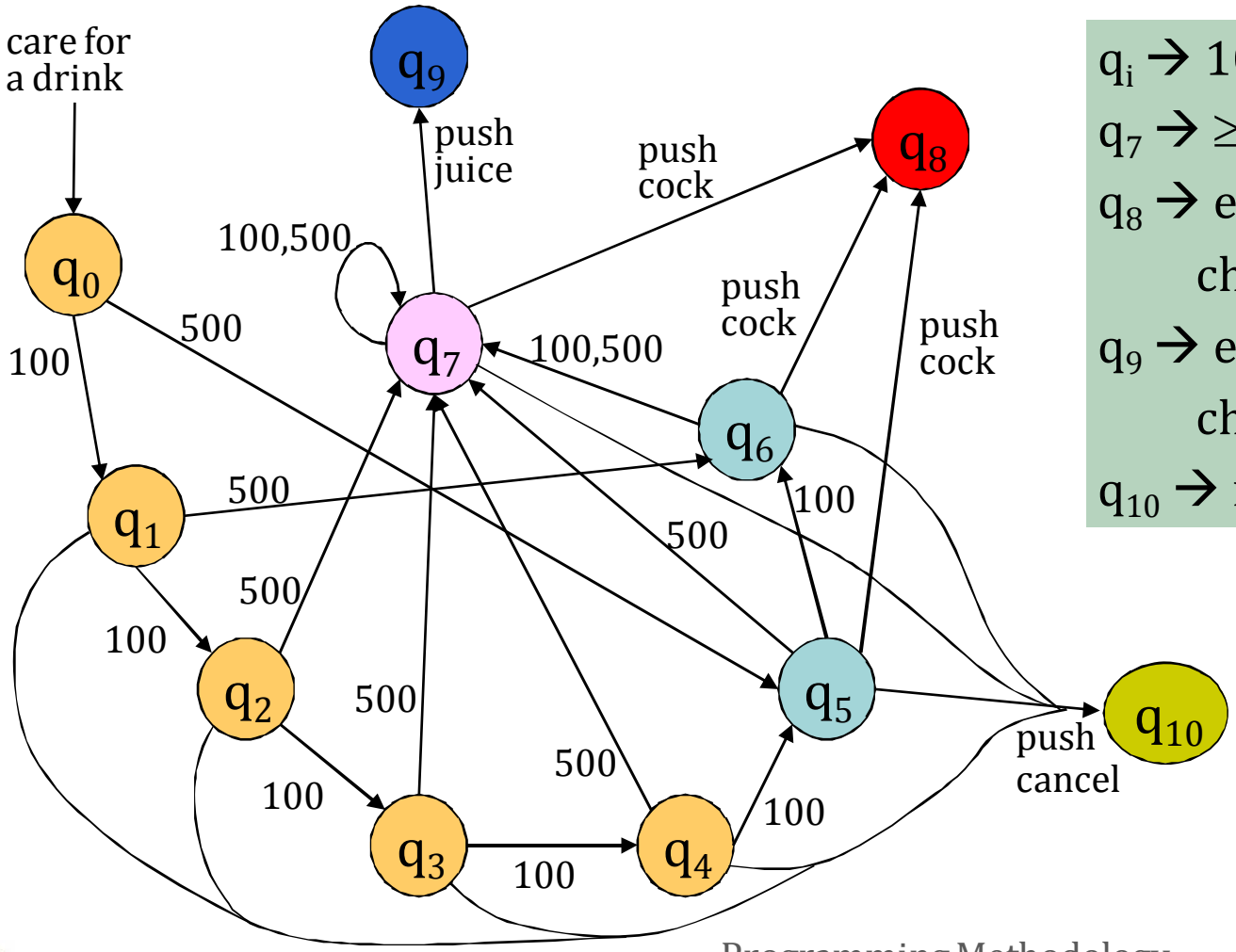
Q = {start, lamp-on, lamp-off}
Σ = {install, push-on-switch, push-off-switch}



Vending machine system

Price: (Coke → 500), (Juice → 700)

Coins to be accepted: 100 and 500



q_i → 100*i Won for $0 \le i \le 6$

q_7 → ≥700 Won

q_8 → eject cock and return change (input – 500)

q_9 → eject juice and return change (input – 700)

q_{10} → return all input money

Building a DFM for this machine will increase the quality of its final S/W code or H/W implementation in terms of ***maintainability***, ***reliability*** *&* ***reusability***.

Topics

Course information

- TAs, course web site, grading, exams, assignments
- Objectives of this course and class schedule

Introduction to programming

- Qualities of software: depending on programming skill
- Software/algorithm specification for programming
- **Programming languages**

Issues that have been covered

- Programming, as an essential component in the software development
- Qualities of software depending on programming skill
  - reliability
  - Efficiency
  - User friendliness
  - Portability
  - Maintainability, Reusability
- Software/algorithm specification for programming
  - descriptive specification: logic and algebraic
  - operational specification: DFM, FSM

Programming languages provide…

- a vehicle for expressing high-level software specification
- a notation for writing algorithms in user code
- a formalism for describing a task to the computer.
  - Like natural languages or software specifications (we just saw), they provide a means for communication between software developers.

  - A well-designed language provides a tool for the user to efficiently instruct/program a machine.

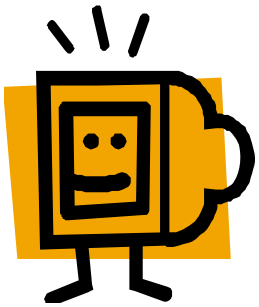
  - If a language is poorly designed/chosen, it may result in …

or even …

Programming languages are…

- an important factor that determine software qualities.
  - Reliability, verifiability - static type checking, aliasing
  - Efficiency: language features for performance, their associated compilers
  - User friendly programming environment
  - Maintainability: simplicity vs. expressiveness, clarity of syntax and naturalness for the application/task, factoring and abstraction
  - Reusability: subroutines, objects
  - Portability: compiler flags or macros (**`#ifdef`**, **`#ifndef`**, **`#define`**, …)
- classified into several paradigms and levels.
  - paradigms: imperative, object-oriented, functional, logic
  - levels: high-level(human oriented), machine-level(machine oriented)
  - Each programming paradigm and level has …
    - its own principles and philosophies in programming.
    - different emphasis on the representative qualities of software.

Programming language classification



Real programming languages

- Numerous programming languages have been developed since the advent of the electronic computer.
  - machine/assembly-level languages
  - high-level languages: Fortran, Lisp, PL/1, Cobol, C, APL, Ada, …
- Each language has its own representative paradigm.
  - C → imperative
  - C++,Java → object oriented
  - Lisp, ML, scheme → functional
  - Prolog → logic
- But often it is very unclear to characterize one language with one paradigm.

  Ex) one language may represent multiple paradigms.
  - C++,Java → object-oriented + imperative
  - scheme → imperative, object-oriented, functional

Machine level languages

- Machine languages
  - In 1940s when the first electronic computer was introduced, all programs were written in machine code.
  - use of numeric codes
    - Ex: "*add 5 and the contents of address 127 and store the result into address 127 and then jump to address 18*"

```
000000001010: 0010 0101 01111111
000000001100: 0101 0000 00010010
                  ⋮
000000010010:     ......
000000010100:     ......
```

  - difficult to read/maintain/modify the codes
    - Ex: right after address 3 insert a new instruction, "*multiply the contents of address 64 by 4 and store the result into address 64*"

```
1011 0100 01000000
```

Machine level languages

- Assembly languages
  - Soon after machine code was used, assembly languages that use *pseudo codes* (translated to machine code by the assembler) were introduced.

Ex:

```
    add 5, I
    jmp L
      `
      `
      `
L:  ......
    ......
```

  - relative addressing
    - Absolute addresses are determined by the linker/loader (not by user)
    - Ex: insert a new instruction "**`mul 4, J`**"
  - easier to read/write a code, but not enough
    - unstructured → difficult to write and maintainable
    - still machine-oriented programming → not portable
    - error-prone and long development time

High-level programming languages

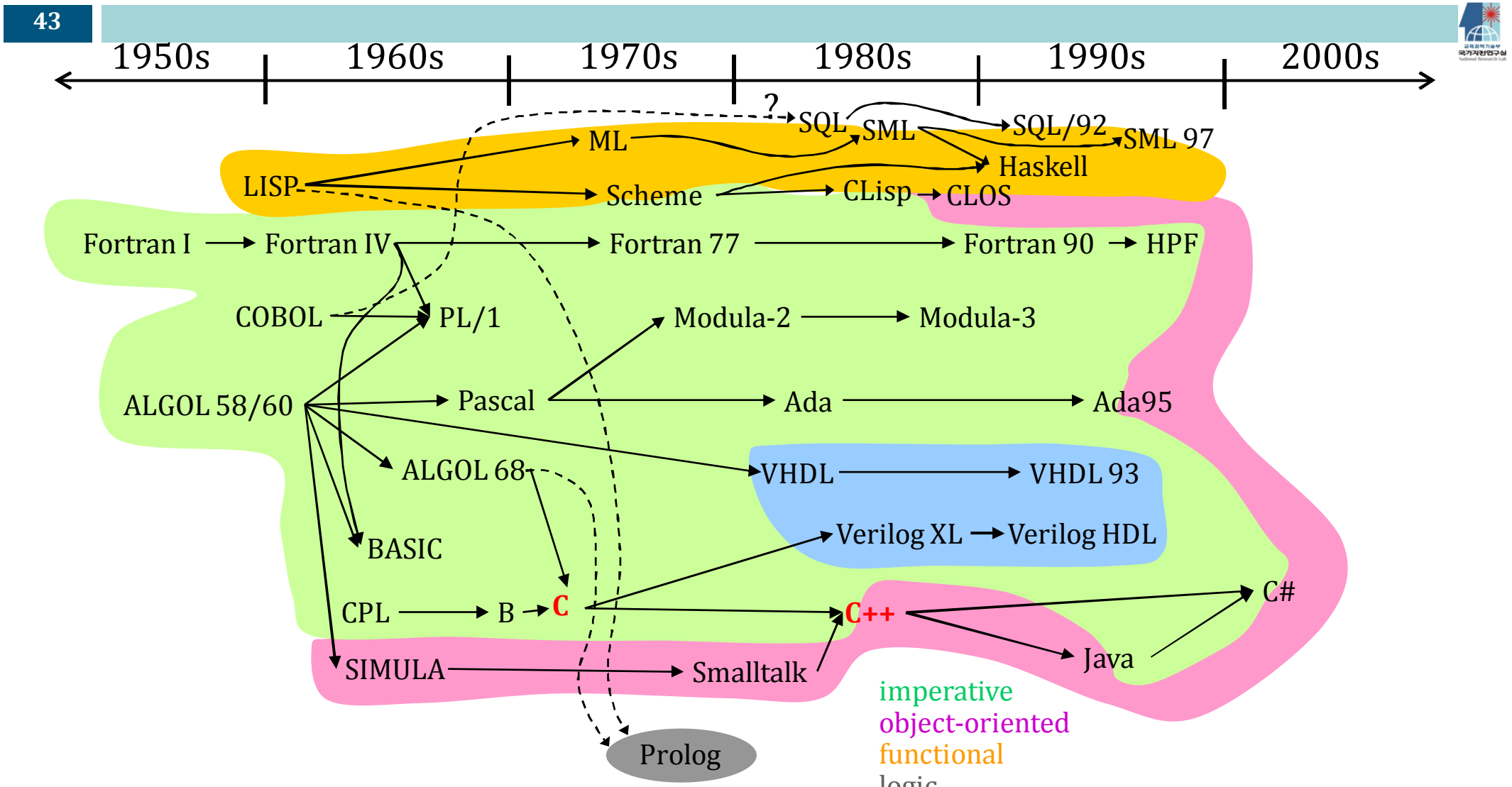
- From mid 40s, primitive forms of interpreted high-level programming languages had been studied.
- In 1954, the first compiled high-level language, Fortran I, was announced for the IBM 704.
  - Numerous high-level languages have been introduced since then…
  - Fortran, C/C++, Lisp, Html, Ada, …
- properties
  - human oriented programming with full of *syntactic sugars*
  - structured, readable, portable, reusable, …
- Ex:

```
    I = I + 1
    goto L
    J = J * 4
    ....
L:  ......
```

→ using high-level mathematical notations

- less efficient?

Genealogy of high-level languages



Genealogy of high-level languages

- First generation languages (late 50s~early 60s)
  - Fortran (for scientific applications), COBOL (for business applications)
  - simple programming structures
    - control structures – non-nested branches, no recursion
    - data structures - primitive structures, static storage management
    - no block structures, no pointers
    - static type checking
  - other early languages
    - APL, SNOBOL - dynamic type checking and storage management
      - used for string or array operations
    - Pure Lisp - pure functional programming, processing of lists (noncontiguous memory cells chained with pointers) for AI

Code examples

- Fortran

```
      subroutine saxpy(n,sa,sx,incx,sy,incy)
C     constant times a vector plus a vector.
      real sx(*),sy(*),sa
      integer i,incx,incy,ix,iy,m,mp1,n
      if(n<=0)return
      if (sa == 0.0) return
      if(incx==1&&incy==1)go to 20
      ix = 1
      iy = 1
      if(incx<0)ix = (-n+1)*incx + 1
      if(incy<0)iy = (-n+1)*incy + 1
      do 10 i = 1,n
        sy(iy) = sy(iy) + sa*sx(ix)
        ix = ix + incx
        iy = iy + incy
   10 continue
      return
   20 m = mod(n,4)
      if( m == 0 ) go to 40
      do 30 i = 1,m
        sy(i) = sy(i) + sa*sx(i)
   30 continue
      if( n < 4 ) return
   40 mp1 = m + 1
      do 50 i = mp1,n,4
        sy(i) = sy(i) + sa*sx(i)
        sy(i + 1) = sy(i + 1) + sa*sx(i + 1)
        sy(i + 2) = sy(i + 2) + sa*sx(i + 2)
        sy(i + 3) = sy(i + 3) + sa*sx(i + 3)
   50 continue
      return
      end
```

- Cobol

```
      *    C O M P U T E   L O A N   A M O U N T    *
       004000-COMPUTE-PAYMENT.
           MOVE 0 TO LW-LOAN-ERROR-FLAG.
           IF (LW-LOAN-AMT ZERO)
               OR
              (LW-INT-RATE ZERO)
               OR
              (LW-NBR-PMTS ZERO)
               MOVE 1 TO LW-LOAN-ERROR-FLAG
               GO TO 004000-EXIT.
           COMPUTE LW-INT-PMT = LW-INT-RATE / 1200
               ON SIZE ERROR
                   MOVE 1 TO LW-LOAN-ERROR-FLAG
                   GO TO 004000-EXIT.
           COMPUTE LW-PMT-AMT ROUNDED =
               (LW-LOAN-AMT * LW-INT-PMT) /
               (1 - 1.00000000 / ( (1 + LW-INT-PMT)
                      ** LW-NBR-PMTS) )
               ON SIZE ERROR
                   MOVE 1 TO LW-LOAN-ERROR-FLAG
                   GO TO 004000-EXIT.
           COMPUTE LW-TOTAL-PMTS = LW-PMT-AMT
                        * LW-NBR-PMTS
               ON SIZE ERROR
                   MOVE 1 TO LW-LOAN-ERROR-FLAG
                   GO TO 004000-EXIT.
           COMPUTE LW-TOTAL-INT = LW-TOTAL-PMTS
                  - LW-LOAN-AMT.
       004000-EXIT.
           EXIT.
```

Genealogy of high-level languages

- Second generation languages (60s)
  - Algol-60, PL/1, Basic
  - block structures – *begin-end* pair
    - control name space and dynamic storage allocation
  - recursive calls - due to dynamic storage management
  - more structured control - *while/for* statements
- Third generation languages (early 70s)
  - Algol-68, Pascal, Simula, C
  - user-defined data structures and types - *struct, record*
  - simple language structures and efficient object code
- Fourth generation languages (70s)
  - Ada, SETL, CLU, Modula-2, Mesa, Gypsy
  - modules, information hiding, data abstraction

Genealogy of high-level languages

- Fifth generation languages (mid 70s~90s)
  - proliferation of programming paradigms
  - functional (or some experimental) programming
    - *Scheme, Common Lisp, Haskell*
    - data base query languages: *SQL*
  - logic programming → *Prolog*
  - object-oriented programming
    - *C++, Smalltalk*
  - imperative programming
    - *Fortran 90*
  - concurrent or parallel programming
    - *High Performance Fortran, Split-C, Concurrent C, Sisal*

Genealogy of high-level languages

- Sixth(?) generation languages (mid 90s~)
  - post-PC era
    - proliferation of embedded and internet systems
  - domain-specific languages
    - *esterel, matlab, DFL, Silage, Numeric-C*
  - architecture description languages
    - *nML, Mimola, Expressions*
  - portable, light-weight languages for internet
    - *Html, Java, C#*
  - etc:
    - aspect-oriented programming
    - meta languages or specification languages: XML, UML, …