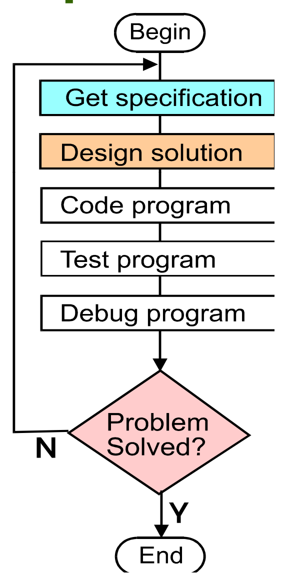
# Topic 2: Specification and design

1. Specification and UML
2. Object-oriented design
3. Modular decomposition
4. Algorithm-design tools
5. Functions and computation

## The system development process

- Development process is iterative
- Specification
  - Understanding the problem
  - input; output; their relationship
  - Unified Modeling Language
- Design
  - Language-independence
  - Module hierarchies, flowcharts, pseudocode
  - Coding, testing come later



## Four CS I skills objectives

- Vague specification → precise specification (analysis)
- Problem specification → flowchart (design)
- Flowchart → trace (design verification)
- Flowchart → program (coding)
- Program → trace (testing)



## 1. Specification and UML

- **Unified Modeling Language:** a standard graphical notation for system specification and design
- Motivated by need to depict interactions between systems and their environments, intiated by external *actors*
- Diagrams include *use-case, activity, class, state, interaction*
- Supports an *object-oriented* methodology

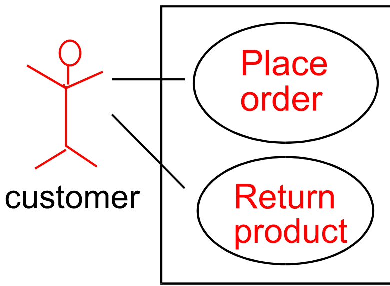
## Software projects start with problem descriptions

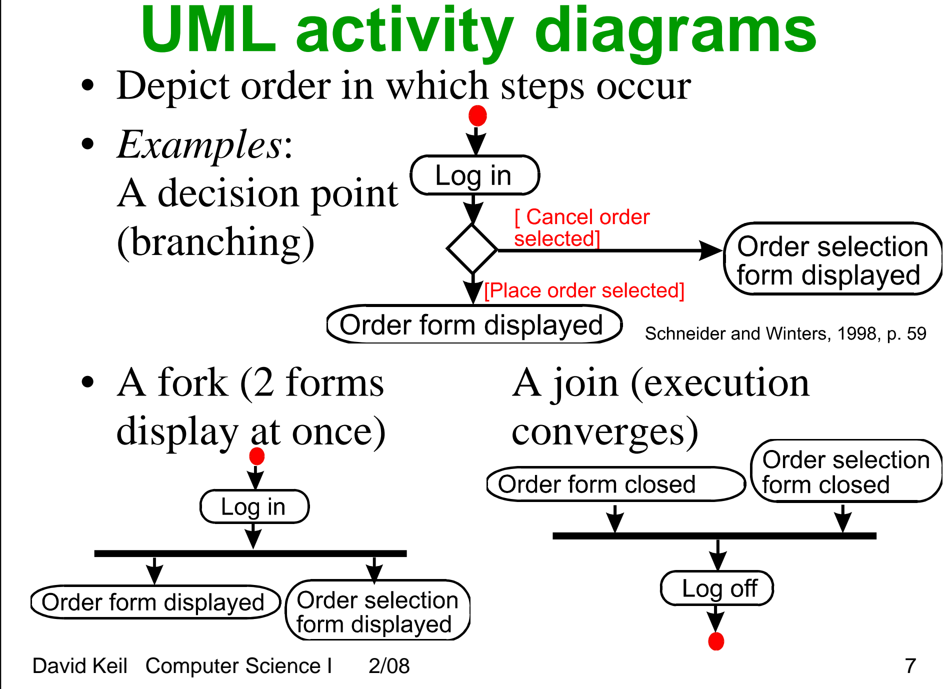
- What service must the software provide?
  *Example:* Process customer orders from catalog
- What assumptions are made?
  *Example:* Some customers may find Web access convenient
- What risks are involved?
  *Example:* Some users are inexperienced with Web access

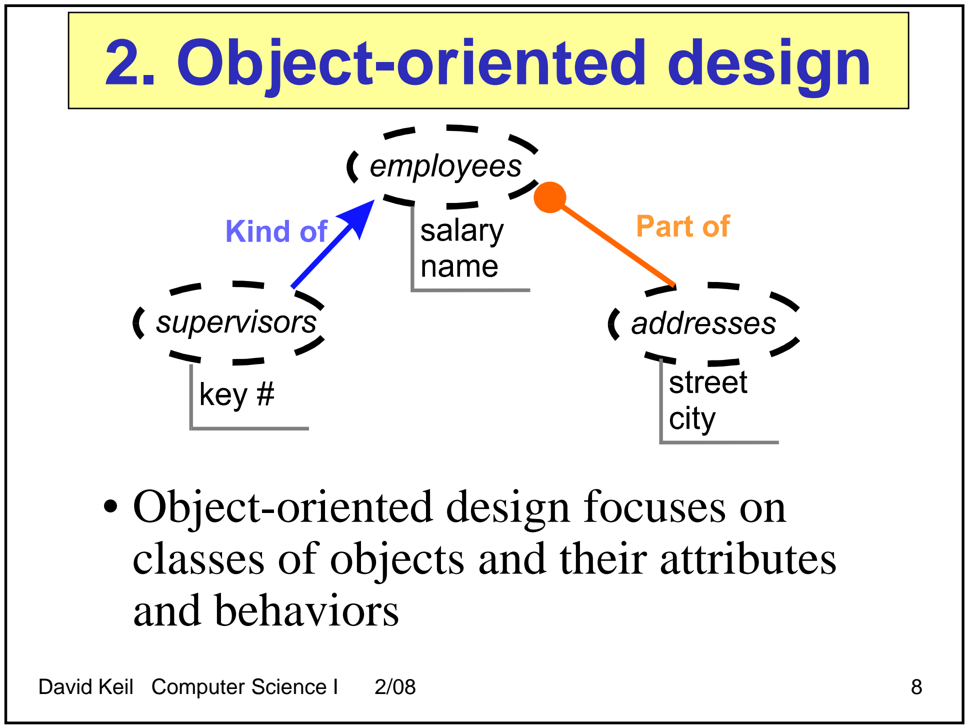


## Use cases and system specification

- Use case: A typical interaction between system and an *actor* in its environment
- Actor: A role, such as customer, manager, supplier, salesperson
- Actors initiate use cases
- Example of two uses cases (UML use-case diagram):




UML activity diagrams
• Depict order in which steps occur
• Examples:
A decision point (branching)
Log in
[ Cancel order selected]
Order selection form displayed
[Place order selected]
Order form displayed
Schneider and Winters, 1998, p. 59
• A fork (2 forms display at once)
Log in
Order form displayed
Order selection form displayed
A join (execution converges)
Order form closed
Order selection form closed
Log off
David Keil Computer Science I 2/08
7


2. Object-oriented design
employees
Kind of
salary
name
Part of
supervisors
key #
addresses
street
city
• Object-oriented design focuses on classes of objects and their attributes and behaviors
David Keil Computer Science I 2/08
8

## The object-oriented paradigm

- *Object:* An instance of a class
- *Class:* A category defined by data *attributes* (properties) and *operations* ("methods")
- In object-oriented programming, objects interact with each other via *messages*
- Objects may *contain* other objects
- Subclasses may *inherit* from other classes
- *Example*: Right-clicking an icon displays the properties and methods of the class of the object the icon represents

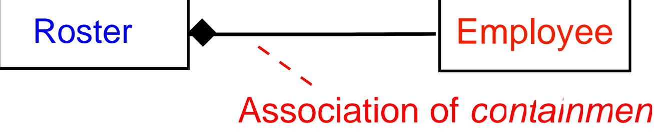


## UML class diagrams

- Rectangle with 3 horizontal compartments

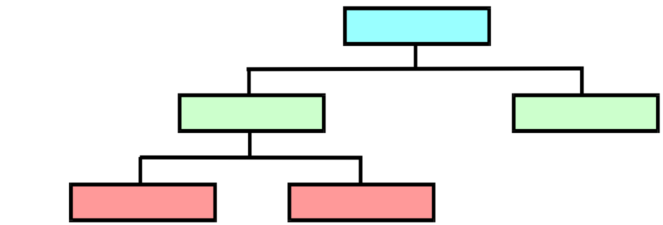
| Employee |
|---|
| name<br>ID<br>salary |
| display()<br>input()<br>calc-paycheck() |

- Class association diagram



# 3. Modular decomposition

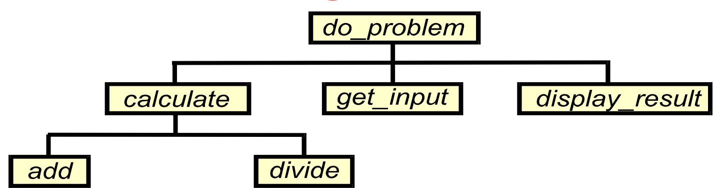
- One strategy: divide and conquer
- All programming languages support modularity
- An organization is modular



- Modular design may be *top-down*
- Subprograms implement modular designs



# Module hierarchy charts tell which subprogram uses which




- Module *do_problem* uses modules *calculate*, *get_input* and *display_result*
- *calculate* uses *add* and *divide*
- Stepwise refinement may lead to further decomposition
- Module hierarchy contains names, not steps, in contrast to flowchart

## Guidelines for writing subprograms

- A module has a single purpose
- Its purpose is documented in a comment at the top
- Code longer than a page usually should be broken down
- Experienced programmers avoid side effects (e.g., output, modification of global variables)

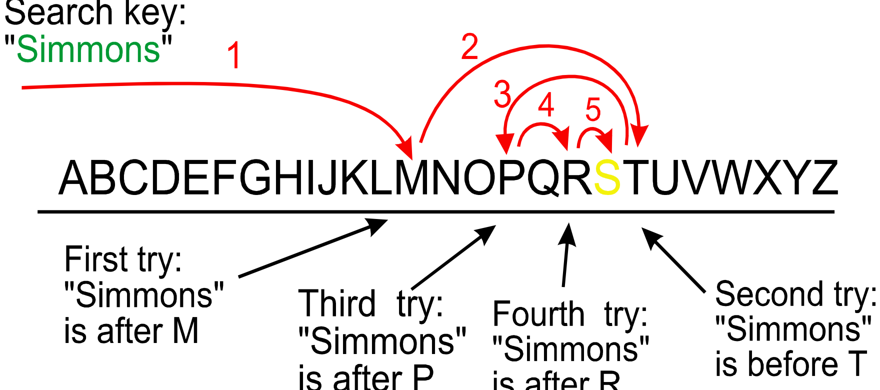


## A divide-and-conquer strategy to solve problem of walking *n* steps

```
Walk (steps)
If steps > 0
   Take a step
   Walk (steps – 1)
```

- This algorithm is *recursive* because it uses itself

## Binary phone-book search




- Each step eliminates half the unsearched data, cuts remaining work in half



## Operator and function notation

- Two possible ways to express the same transformation of data
- *Examples:*

| | |
|---|---|
| $a + b$ | $sum(a,b)$ |
| $(2 + 5) \times 4$ | $product(sum(2,5), 4)$ |
| $\sum_{k=1}^{4} k$ | $sum(1,2,3,4)$ |

- Here the name of a *function* could also be the name of a *module, subprogram*, or *Java method*

## 4. Algorithm design tools

- Both notations show order of execution
- Pseudocode
  - informal
  - precise
  - text outline format
- Flowcharts
  - graphical
  - shapes denote steps
  - arrows show flow of control

Begin/end
Processing
I/O
Decision test
Flow

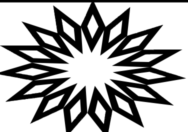


## Algorithm:

A precise plan to solve a problem in a finite number of steps

- Program designs use algorithms
- Most computation is algorithmic
- Flowcharts and pseudocode can express algorithms

# Algorithmic thinking



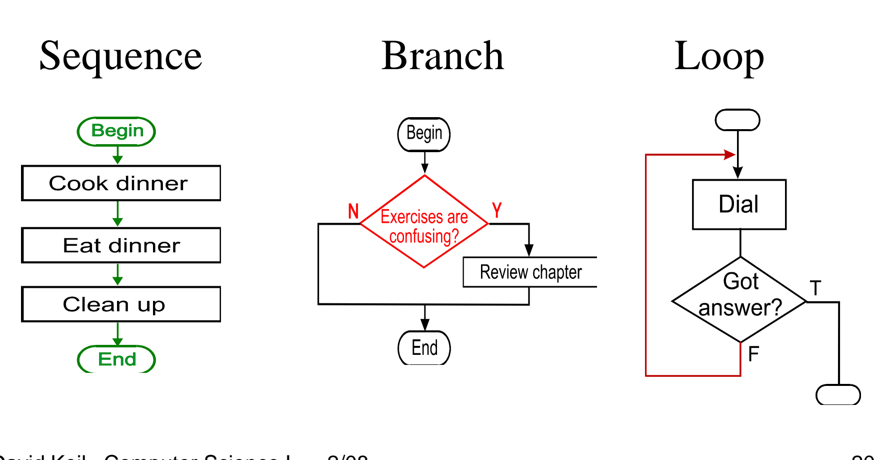
- *Definition:* an algorithm is a step-by-step plan to transform input to output in a finite number of steps
- Essential properties (L. Snyder):
  - Specification of input, output, relationship between them
  - Definiteness (deterministic sequence of operations)
  - Effectiveness (doability)
  - Finiteness



# Control structures

All algorithms may be built from three basic *control structures*:

Sequence Branch Loop



## The *sequence* control structure

*Problem:*
Accomplish dinner routine

*Pseudocode:*

1. Cook dinner
2. Eat dinner
3. Clean up

*Flowchart:*

Begin → Cook dinner → Eat dinner → Clean up → End



## An algorithm to add numbers

1. Prompt for integers *input1, input2*
2. *sum* ← *input1* + *input2*
3. Display *sum*

- In pseudocode and flowcharts, the symbol ← stands for assignment of a value to a variable
- An algorithm is almost always *general purpose*: it works with *variables*, not only specific constant values

## The *decision* control structure

*Problem:*
Prepare to do homework

*Pseudocode:*
If exercises are confusing
  review chapter

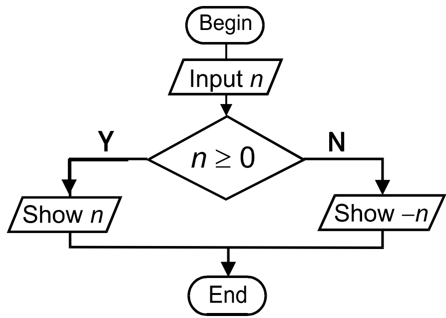
*Flowchart:*

Begin
Exercises are confusing?
N
Y
Review chapter
End



## Finding absolute value

**Input *n***
**If $n \geq 0$**
  **display *n***
**otherwise**
  **display (–*n*)**

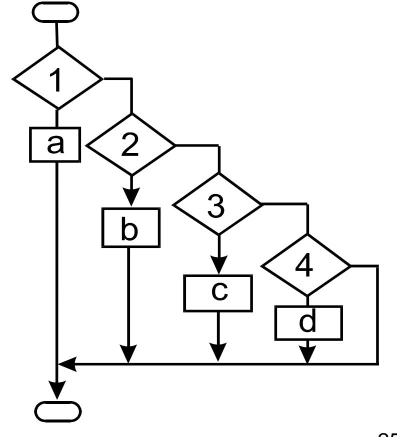



- With the branch control structure, one and only one of the alternatives executes
- In pseudocode, the subordinate (con-ditional) steps are normally indented

# Multiple alternatives

- Require multiple diamonds in flowchart

**If *n* = 1 display ‘a’**
**otherwise**
**if n = 2 display ‘b’**
**otherwise**
**if n = 3 display ‘c’**
**otherwise**
**if n = 4 display ‘d’**



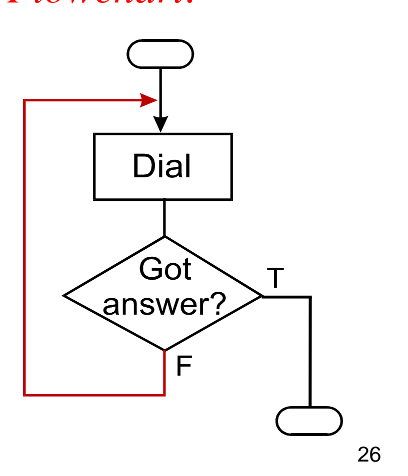


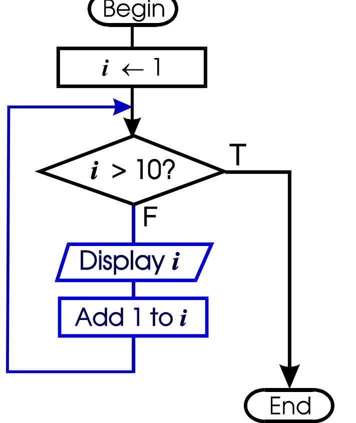

# The *repetition* (loop) control structure

*Problem:*
Telephone someone

*Pseudocode:*
Repeat
Dial number
until someone answers

*Flowchart:*

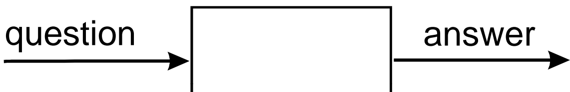





**Problem:** Display all values from 1 to 10

- A *counted* loop; counter is *i*

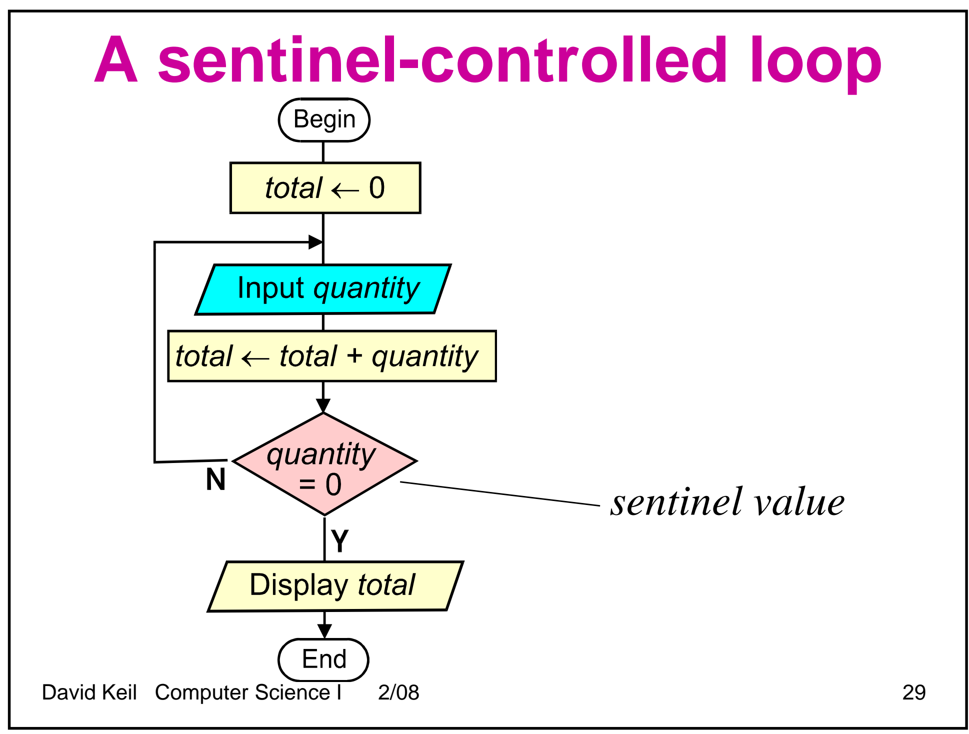





or process

question → [ ] → answer

- Natural language
- Design notation
  - Flowchart
  - Pseudocode
  - Unified Modeling Language (UML)
- Programming language (JavaScript, Java, C++…)




# Tracing an algorithm

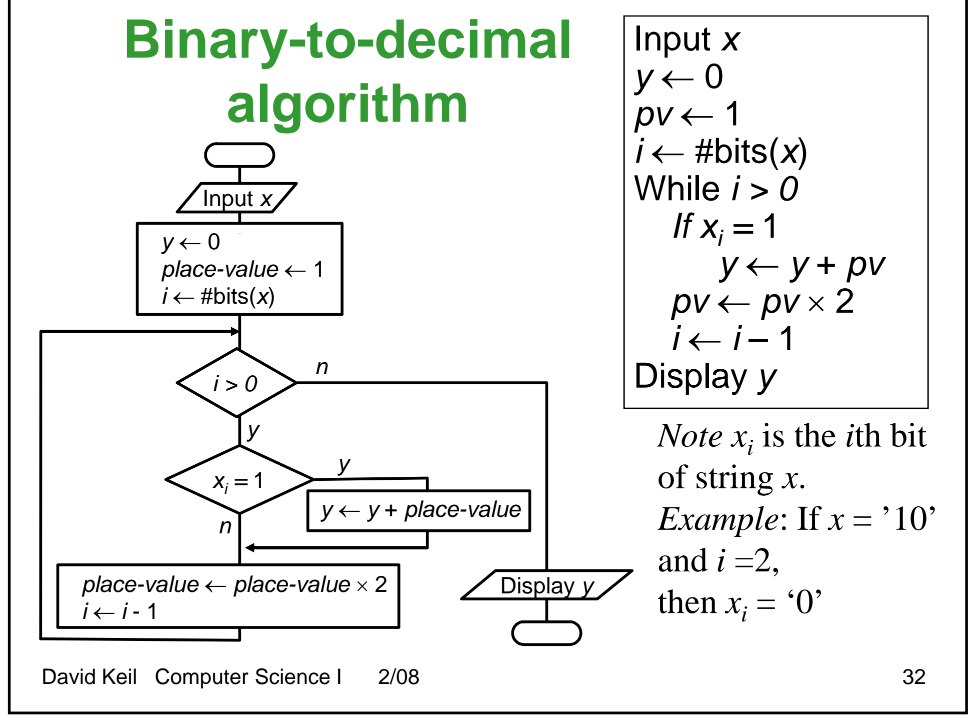
- Allows designer to check result of algorithm, including internal (undisplayed) values
- Use one column per value traced; one row per loop iteration.
- *Example* (See previous slide):

| *quantity* | *total* | *output* |
|---|---|---|
| | 0 | |
| 3 | 3 | |
| 2 | 5 | |
| 1 | 6 | |
| 0 | 6 | 6 |

## Problem: binary to decimal

- *Problem:* Convert binary string to decimal value
- *Analysis:* Recall (Topic 4) that a numeral represents the *sum* of *place values*, which are each $b^i$, where *b* is base (2, 10, …) and *i* is the number of places from the right
- *Solution* (see next slide):
  - Let *x* be input bit sequence, *y* be output value
  - Start at rightmost bit (counter *i* = # bits)
  - Loop through bits, R to L
  - At each 1-bit, add a power of 2 (place-value)
  - Double place-values, decrement *i* at each iteration of loop





## Loop invariants

- A loop invariant is a logical (true/false) assertion about values and outputs at the beginning of a loop iteration
- *Example*
  Loop invariants for the previous slide are
  - The value of $i$ is between 0 and 11
  - All values in $\{1 \ldots i - 1\}$ have been displayed
- Loop invariants help assure correctness



## Three control structures are sufficient

- Three control structures: *sequence, branch*, and *loop*
- They are used in structured programming; may be combined
- *Any computable problem* can be solve using only these three
- *Note:* Some problems are not computable

## Combining, nesting control structures

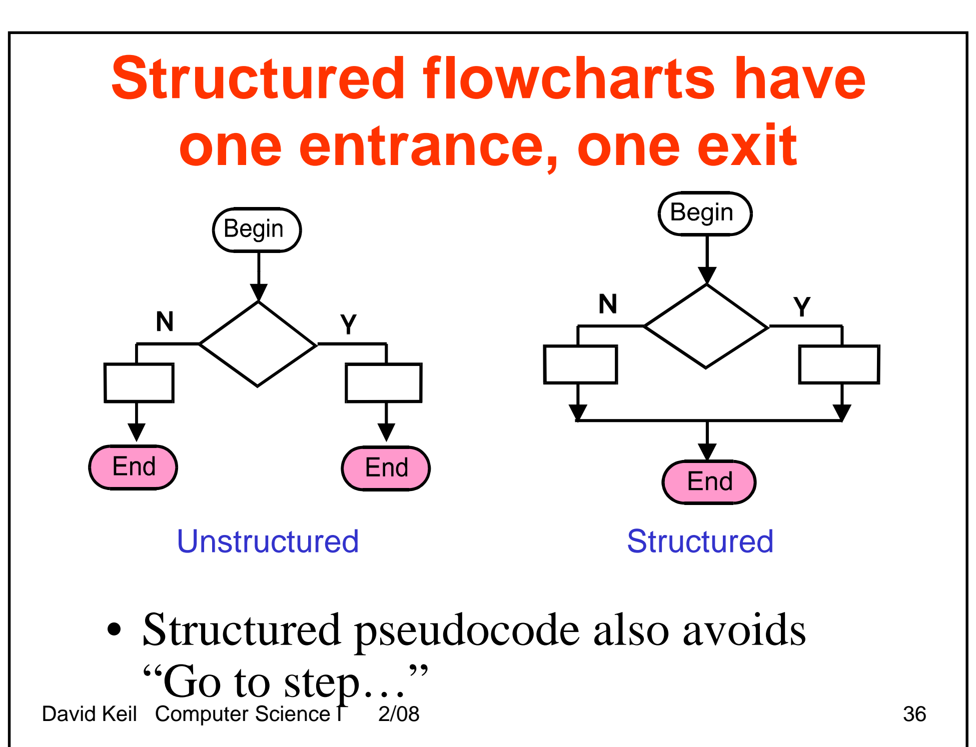
Begin
Input *launches*
*launches* > 0?
F
T
$i \leftarrow 10$
Display $i$
$i \leftarrow i - 1$
T
$i > 0$?
F
*launches* ← *launches* – 1
End

*Loop within loop*

Begin
$i \leftarrow 100$
Input *height*
*height* > 72?
T
Display "Tall"
F
$i \leftarrow i - 1$
T
$i > 0$?
F
End

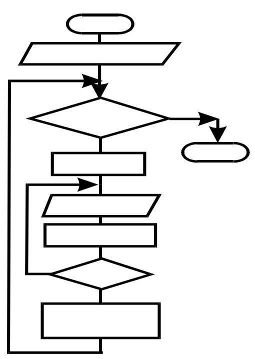
*Branch within loop*

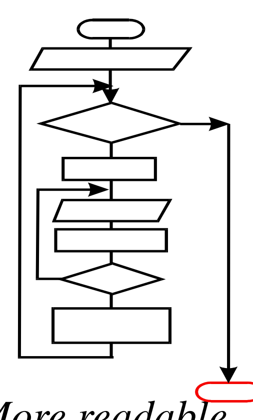




## The picture tells a story

- A flowchart should show what happens, at a glance





*Not too clear* *More readable*



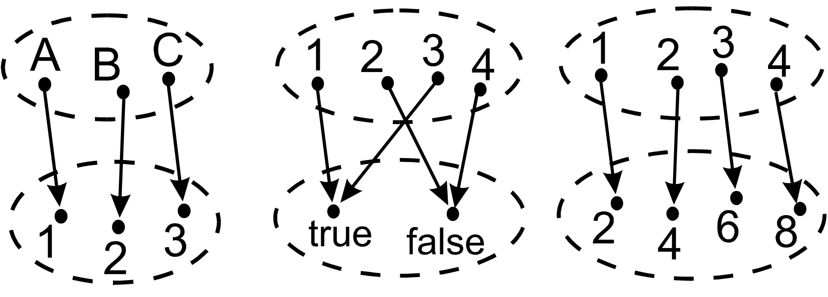
## 5. Functions and computation

- A function in mathematics is a passive *set* of *pairs* that maps sets
- A computation entails *action*
- A computer program or subprogram computes a mathematical function
- A C or C++ “function” is actually a subprogram, not a mathematical function

# Review of functions

- *Function:* A set that is a mapping from one set to itself or to another set
- *Examples:*

*Index*('B') = 2 *Odd* (3) = *true* *Twice*(1) = 2






# Computable functions

- A mapping obtained by an effective step-by-step procedure
- *Examples:* successor, sum
- Computer programming is concerned with coding computable functions
- An uncomputable function: "Does this program work correctly?"
- The computable functions are the same as the ones that are recursively definable

## Recursive functions and computable functions

- For natural numbers *a, b* in {0, 1, 2, 3…}:

$$sum(a,b) = \begin{cases} b & \text{if } a = 0 \\ sum(a-1, b+1) & \text{otherwise} \end{cases}$$

- The definition of *sum* is *recursive* because it uses itself
- The form of this definition is a *recurrence*
- It guarantees computability



## Some recurrences

$$sum(a,b) = \begin{cases} b & \text{if } a = 0 \\ sum(a-1, b+1) & \text{otherwise} \end{cases}$$

$$product(a,b) = \begin{cases} 0 & \text{if } b = 0 \\ a & \text{if } b = 1 \\ a + product(a, b-1) & \text{otherwise} \end{cases}$$

$$even(a) = \begin{cases} true & \text{if } a = 0 \\ false & \text{if } a = 1 \\ even(a-2) & \text{otherwise} \end{cases}$$

## Russian peasants' algorithm

***Product (a, b)***
*result* ← 0
while *a* > 0
  if *a* is odd
    add *b* to *result*
  $a \leftarrow \lfloor a \div 2 \rfloor$
  $b \leftarrow b \times 2$
Return *result*

*Example:* $13 \times 5 = 65$

| | *a* | *b* | *product* |
|---|---|---|---|
| | | | 0 |
| 1. | 13 | 5 | 5 |
| 2. | 6 | 10 | 5 |
| 3. | 3 | 20 | 25 |
| 4. | 1 | 40 | 65 |
| 5. | 0 | 80 | |



## Non-algorithmic computations

- The process diagrammed at right will never terminate if user continues to input a non-zero value
- Interaction, alternating input and output is not algorithmic,

## Generations of programming languages

- First generation: Machine languages (binary)
- 2nd generation: Assembler languages, processor specific
- 3rd generation: Procedural, high-level, hardware independent (C, BASIC, JavaScript)
- 4th generation: Nonprocedural query or report-generation languages (SQL, RPG)
- Declarative languages (Prolog)
- Object-oriented languages (Java)
- Functional languages (Lisp)



## References

Cay Horstmann. *Big Java,* 3rd ed. Wiley, 2008.

J. Parsons and D. Oja. *Computer Concepts,* 9th ed. Thomson, 2007.

L. Snyder. *Fluency with Information Technology.* Addison Wesley, 2006.

G. Schneider and J. Winters. *Applying Use Cases: A Practical Guide.* Addison Wesley, 1998.