CS 242

2012

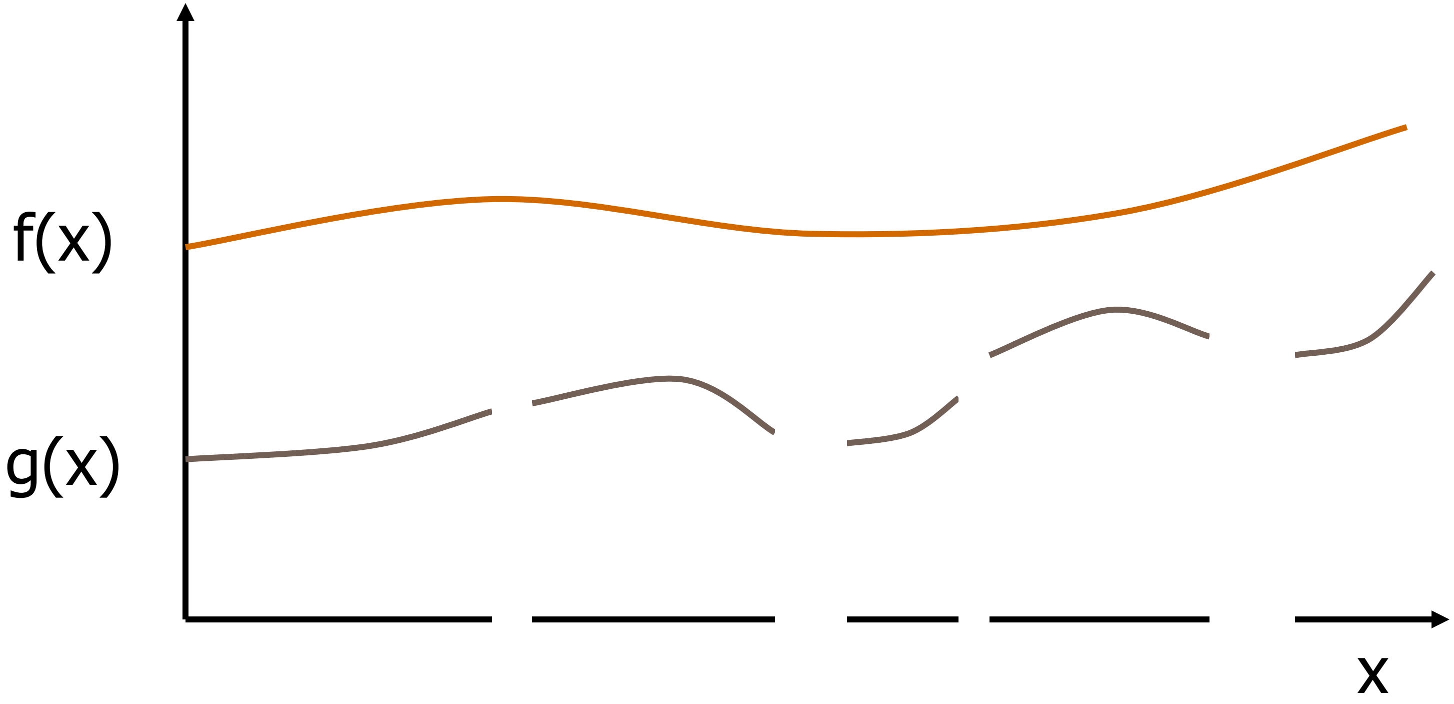
Computable Functions

Reading: Chapter 2

Foundations: Partial,Total Functions

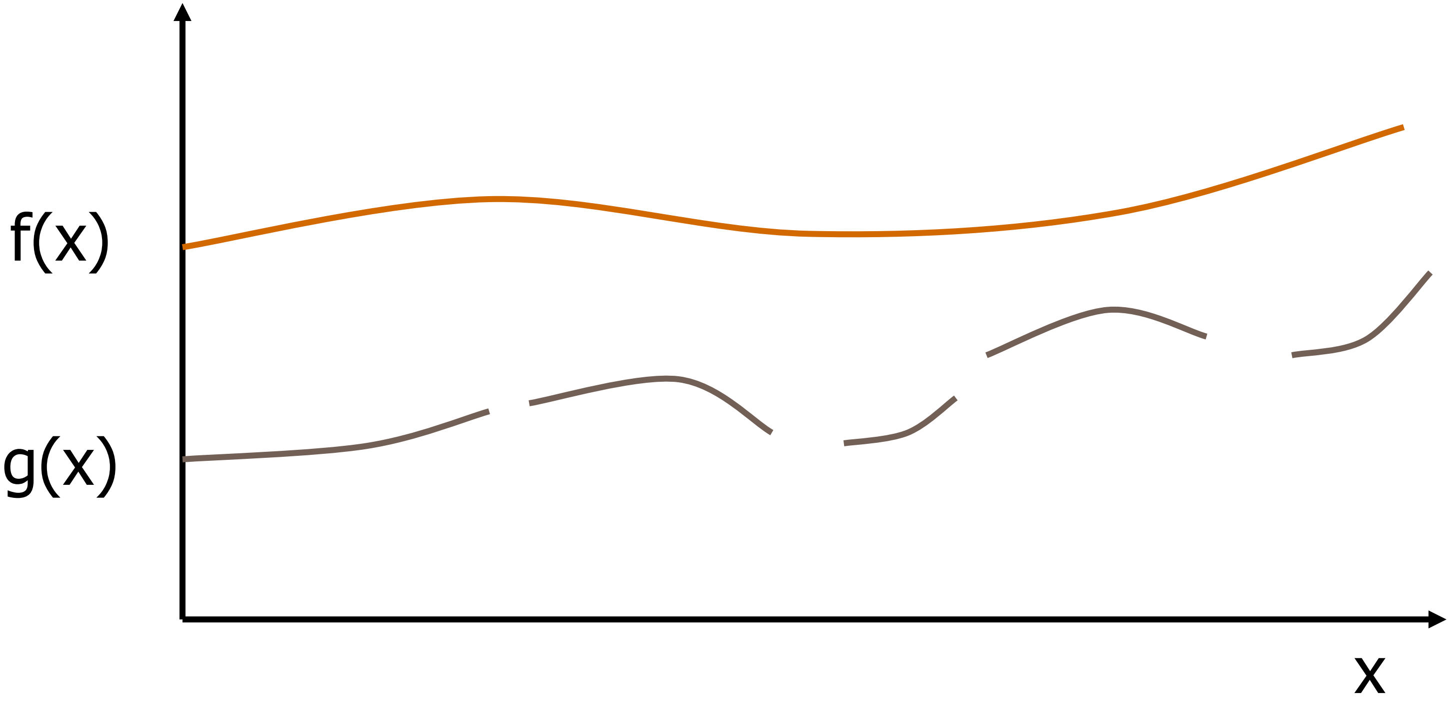
- Value of an expression may be undefined
  - Undefined operation, e.g., division by zero
    - 3/0 has no value
    - implementation may halt with error condition
  - Nontermination
    - f(x) = if x=0 then 1 else f(x-2)
    - this is a *partial* function: not defined on all arguments
    - cannot be detected at compile-time; this is halting problem
  - These two cases are
    - “Mathematically” equivalent
    - Operationally different

Partial and Total Functions



- Total function: f(x) has a value for every x
- Partial function: g(x) does not have a value for every x

Functions and Graphs



- Graph of f = { ⟨x,y⟩ | y = f(x) }
- Graph of g = { ⟨x,y⟩ | y = g(x) }

Mathematics: a function is a set of ordered pairs (graph of function)

Partial and Total Functions

- Total function $f:A\to B$ is a subset $f \subseteq A\times B$ with
  - For every $x\in A$, there is some $y\in B$ with $\langle x,y\rangle \in f$ (total)
  - If $\langle x,y\rangle \in f$ and $\langle x,z\rangle \in f$ then $y=z$ (single-valued)

- Partial function $f:A\to B$ is a subset $f \subseteq A\times B$ with
  - If $\langle x,y\rangle \in f$ and $\langle x,z\rangle \in f$ then $y=z$ (single-valued)

- Programs define partial functions for two reasons
  - partial operations (like division)
  - nontermination

    f(x) = if x=0 then 1 else f(x-2)

Computability

- Definition

  Function f is computable if some program P computes it:

  For any input x, the computation P(x) halts with output f(x)

- Terminology

  Partial recursive functions
  = partial functions (int to int) that are computable

- Church-Turing Hypothesis

  The programming language doesn't matter –
  all "reasonable" programming languages
  define the same class of computable functions

Halting function

- Decide whether program halts on input
  - Given program P and input x to P,

$$\mathit{Halt}\ (P,x) = \begin{cases} \text{yes} & \text{if } P(x) \text{ halts} \\ \text{no} & \text{otherwise} \end{cases}$$

Clarifications

- Assume program P requires one string input x
- Write P(x) for output of P when run in input x
- Program P is string input to *Halt*
- Represent two inputs P, x as string P$x (for example)

Theorem: There is no program for *Halt*

Unsolvability of the halting problem

- Suppose P solves variant of halting problem

  On input Q, assume

  $$P(Q) = \begin{cases} \text{yes} & \text{if } Q(Q) \text{ halts} \\ \text{no} & \text{otherwise} \end{cases}$$

- Build program D

  $$D(Q) = \begin{cases} \text{run forever} & \text{if } Q(Q) \text{ halts} \\ \text{halt} & \text{if } Q(Q) \text{ runs forever} \end{cases}$$

- Does this make sense? What can D(D) do?
  - If D(D) halts, then D(D) runs forever.
  - If D(D) runs forever, then D(D) halts.
  - *CONTRADICTION:* program P must not exist.

Examples

- Is there an algorithm to decide whether this program has a run-time type error?

  ```
  if   f(x)   then  y=1+”Bob” else  y=2+”Alice”
  ```

- Is there an algorithm to decide whether this program reads variable z ?

  ```
  if   f(x)   then  y=z+”Bob”   else  y=z+”Alice”
  ```

Main points about computability

- Some functions are computable, some are not
  - Halting problem
  - Other problems that are equivalent

- Programming language implementation
  - *Can* report error if program result is undefined due to division by zero, other error condition
  - *Cannot* warn user if program will not terminate
  - *Many* useful program properties are *not* computable

CS 242 2012

Data Abstraction and Modularity

Reading: Sections 9.1, 9.2 (except 9.2.5), and 9.3.1

Topics

- Modularity
  - Interface, specification, and implementation
- Modular program development
  - Step-wise refinement ; Prototyping ; …
- Language support for modularity
  - Procedural abstraction
  - Abstract data types
    - Representation independence
    - Datatype induction
  - Packages and modules
  - Generic abstractions
    - Functions and modules with type parameters

Modularity: Basic Concepts

- Component
  - Meaningful program unit
    - Function, data structure, module, …
- Interface
  - Types and operations defined within a component that are visible outside the component
- Specification
  - Intended behavior of component, expressed as property observable through interface
- Implementation
  - Data structures and functions inside component

Example: Function Component

- Component
  - Function to compute square root
- Interface
  - float sqroot (float x)
- Specification
  - If x>0, then sqrt(x)*sqrt(x) ≈ x.
- Implementation

```
float sqroot (float x){
  float y = x/2; float step=x/4; int i;
  for (i=0; i<20; i++){if ((y*y)<x) y=y+step; else y=y-step; step = step/2;}
  return y;
}
```

Example: Data Type

- Component
  - Priority queue: data structure that returns elements in order of decreasing priority
- Interface
  - Type pq
  - Operations empty : pq
    insert : elt * pq → pq
    deletemax : pq → elt * pq
- Specification
  - Insert add to set of stored elements
  - Deletemax returns max elt and pq of remaining elts

Philosophy

- Build reusable program components
- Construct systems by divide-and-conquer
  - Limit interactions between components
  - Each component is assumed to satisfy spec
    - If another component satisfies the same specification, you can replace the first by the second
    - Internal improvements only improve the overall system, not break it

Example program using component

- Priority queue: structure with three operations

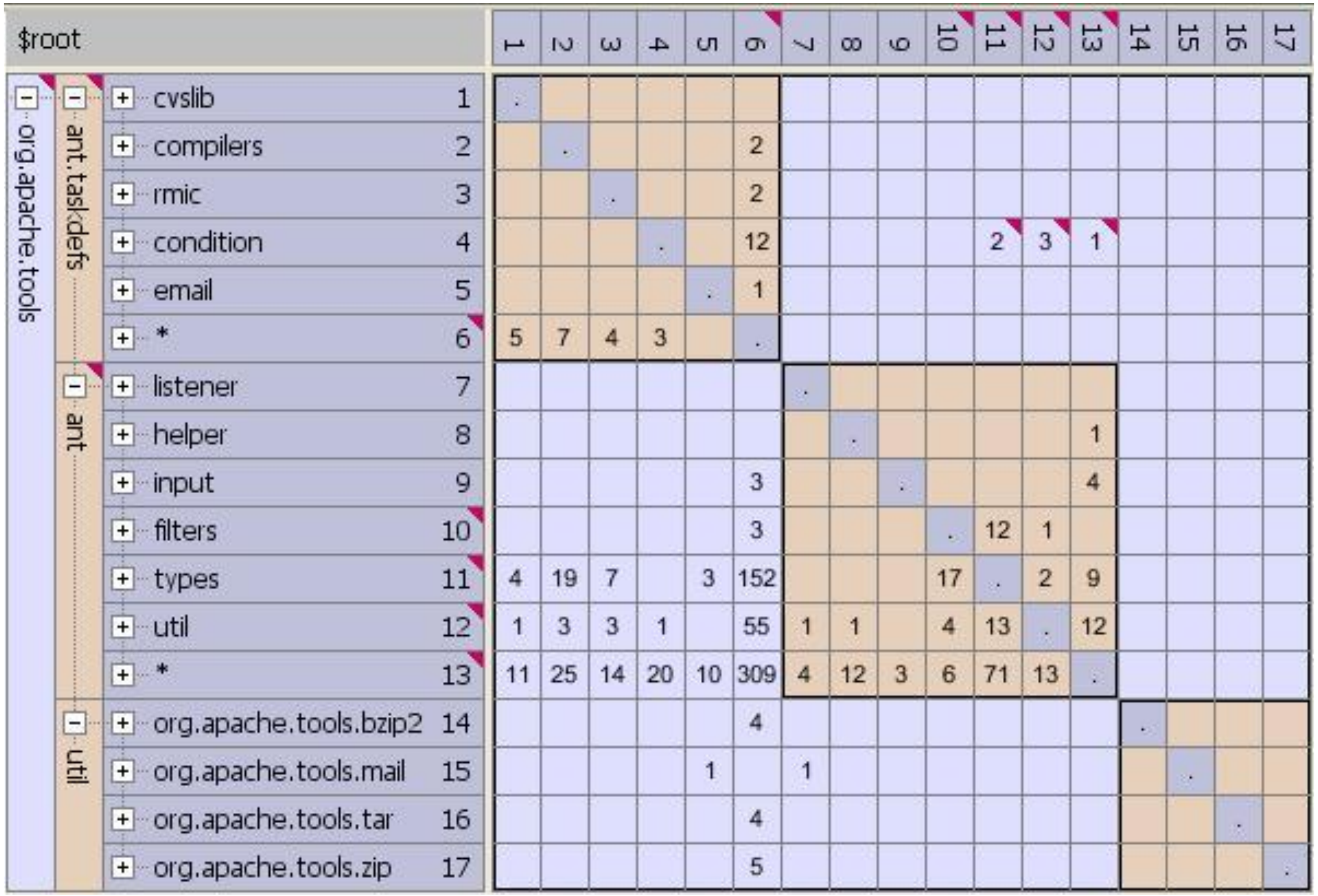
  empty : pq
  insert : elt * pq → pq
  deletemax : pq → elt * pq

- Sorting algorithm using priority queue

```
begin
  create empty pq s
  insert each element from array into s
  remove elements in decreasing order and place in array
end
```

This gives us an O(n log n) sorting algorithm (HW ?)

Component Dependencies



$root				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
org.apache.tools	ant.taskdefs	cvslib	1	.																
		compilers	2		.				2											
		rmic	3			.			2											
		condition	4				.		12					2	3	1				
		email	5					.	1											
		*	6	5	7	4	3		.											
	ant	listener	7							.										
		helper	8								.					1				
		input	9						3			.				4				
		filters	10						3				.	12	1					
		types	11	4	19	7		3	152				17	.	2	9				
		util	12	1	3	3	1		55	1	1		4	13	.	12				
		*	13	11	25	14	20	10	309	4	12	3	6	71	13	.				
	util	org.apache.tools.bzip2	14						4								.			
		org.apache.tools.mail	15					1		1								.		
		org.apache.tools.tar	16						4										.	
		org.apache.tools.zip	17						5											.

source: Lattix.com

Modular program design

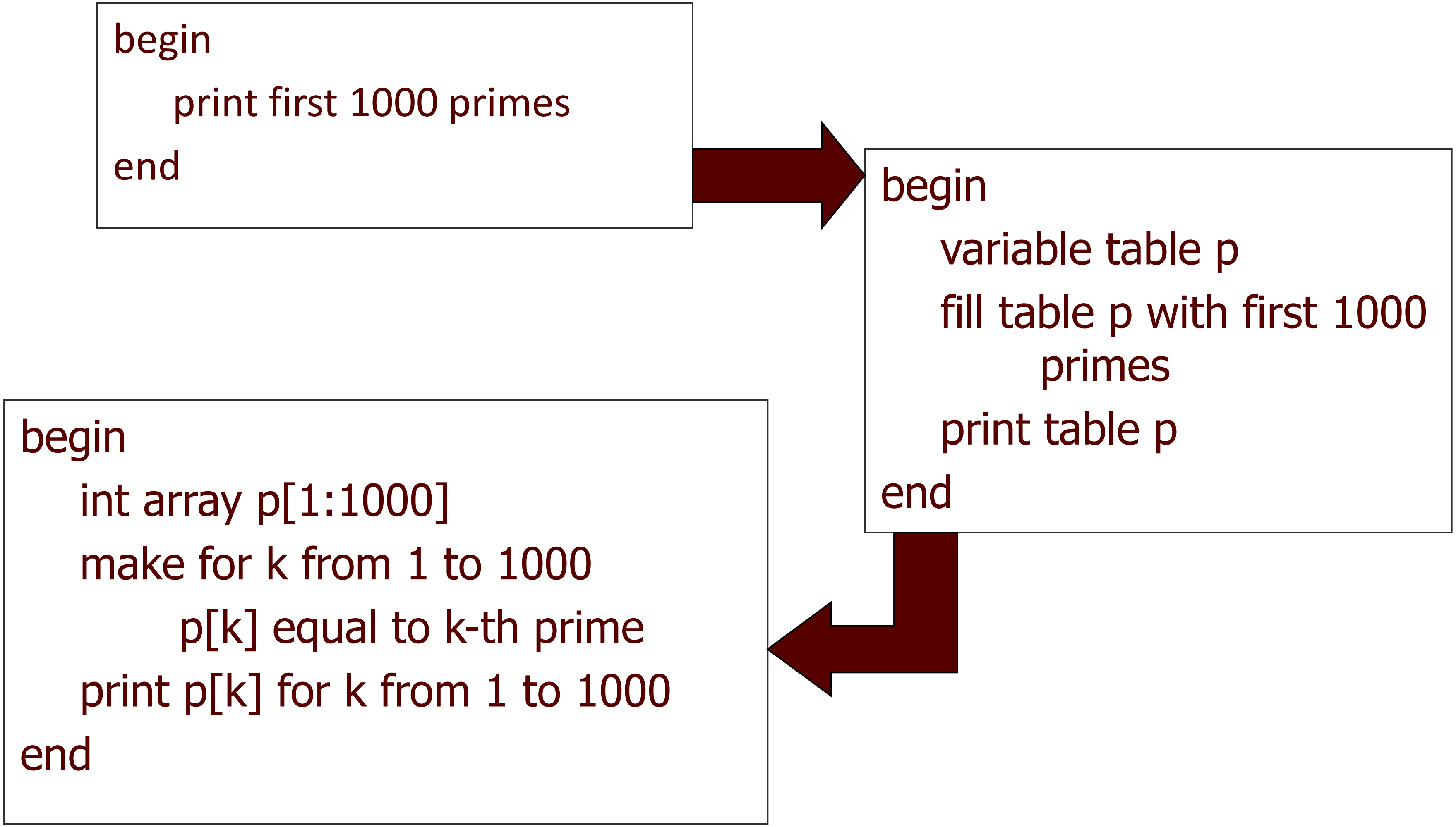
- Top-down design
  - Begin with main tasks, successively refine
- Bottom-up design
  - Implement basic concepts, then combine
- Prototyping
  - Build coarse approximation of entire system
  - Successively add functionality

Stepwise Refinement

- Wirth, 1971
  - "… program ... gradually developed in a sequence of refinement steps"
  - In each step, instructions … are decomposed into more detailed instructions.

- Historical reading on web (CS242 Reading page)
  - N. Wirth, Program development by stepwise refinement, *Communications of the ACM,* 1971
  - D. Parnas, On the criteria to be used in decomposing systems into modules, *Comm ACM,* 1972
  - Both *ACM Classics of the Month*

Dijkstra’s Example (1969)

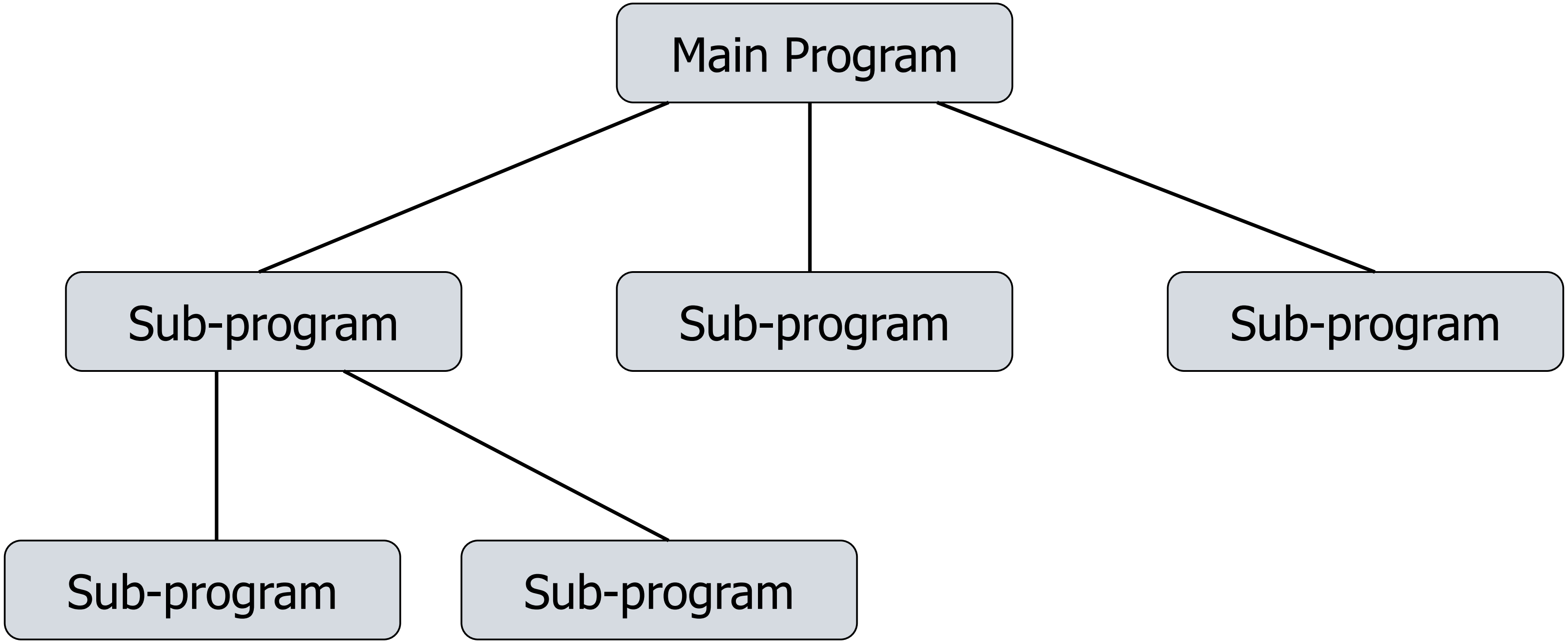


```
begin
    print first 1000 primes
end
```

```
begin
    variable table p
    fill table p with first 1000 primes
    print table p
end
```

```
begin
    int array p[1:1000]
    make for k from 1 to 1000
        p[k] equal to k-th prime
    print p[k] for k from 1 to 1000
end
```

Program Structure


Main Program

Sub-program

Sub-program

Sub-program

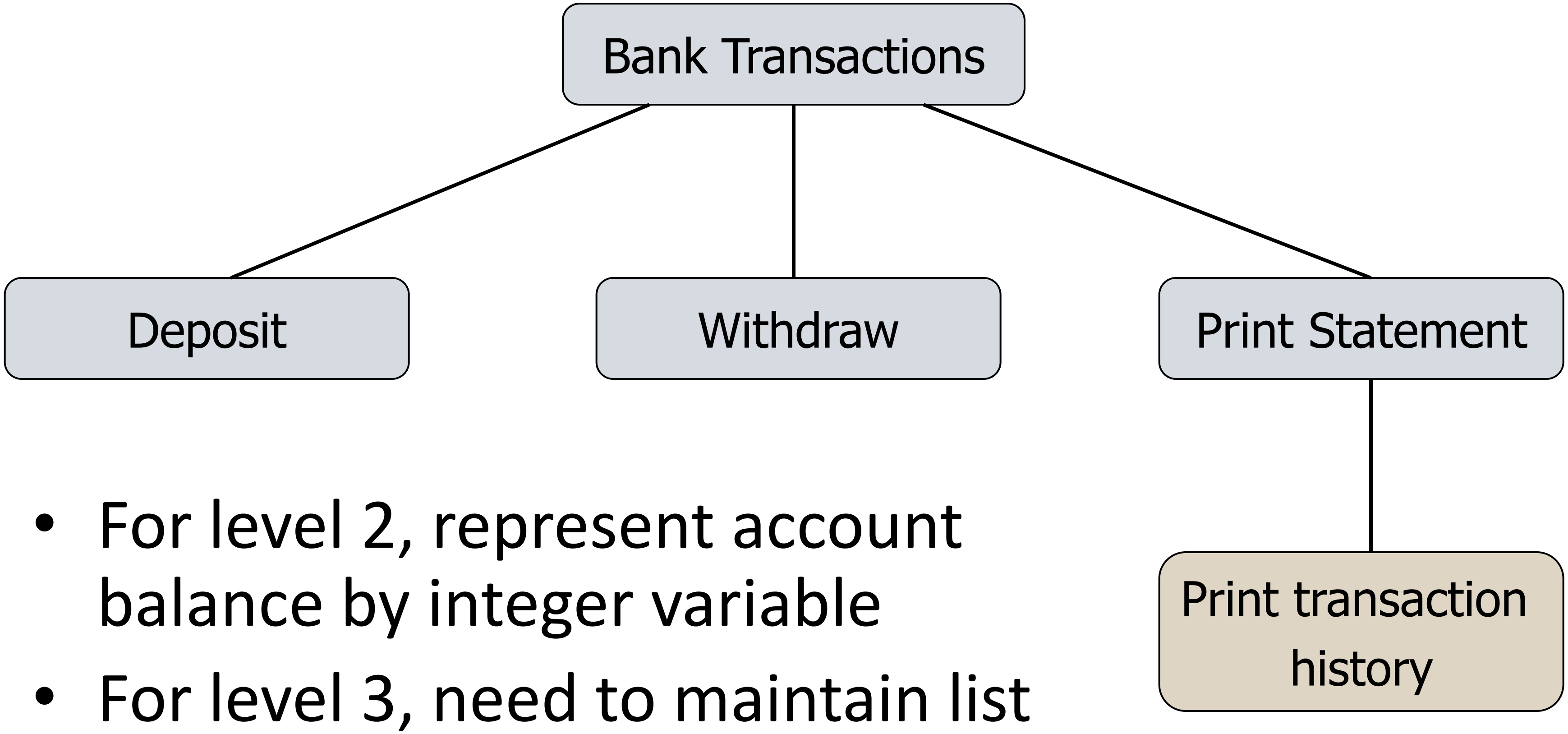
Sub-program

Sub-program

Data Refinement

- Wirth, 1971 again:
  - As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine program and data specifications in parallel

Example



Bank Transactions

Deposit

Withdraw

Print Statement

Print transaction history

- For level 2, represent account balance by integer variable
- For level 3, need to maintain list of past transactions

Language support for modularity

- Interface definition
  - Interface may consist of types, functions, subtype relationships, other language concepts exposed to other modules
- Isolation
  - Restrict dependence to factors visible through explicitly defined interface

Examples

- Procedural abstraction
  - Hide functionality in procedure or function
- Data abstraction
  - Hide decision about representation of data structure and implementation of operations
  - Example: priority queue can be binary search tree or partially-sorted array

Abstract Data Types

- Prominent language development of 1970’s
- Main ideas:
  - Separate interface from implementation
    - Example:
      - Sets have empty, insert, union, is_member?, …
      - Sets implemented as … linked list …
  - Use type checking to enforce separation
    - Client program only has access to operations in interface
    - Implementation encapsulated inside ADT construct

ML Abstype

- Declare new type with values and operations

```
abstype t = <tag> of <type>
  with
    val <pattern> =  <body>
      ...
    fun f(<pattern>) =  <body>
      ...
  end
```

- Representation

`t = <tag> of <type>`   similar to ML datatype decl

Abstype for Complex Numbers

- Input

```
abstype cmplx = C of real * real with
    fun cmplx(x,y: real) = C(x,y)
    fun x_coord(C(x,y)) = x
    fun y_coord(C(x,y)) = y
    fun add(C(x1,y1), C(x2,y2)) = C(x1+x2, y1+y2)
end
```

- Types (compiler output)

```
type cmplx
val cmplx = fn : real * real -> cmplx
val x_coord = fn : cmplx -> real
val y_coord = fn : cmplx -> real
val add = fn : cmplx * cmplx -> cmplx
```

Abstype for finite sets

- Declaration

```
abstype 'a set = SET of 'a list with
    val empty = SET(nil)
    fun insert(x, SET(elts)) = ...
    fun union(SET(elts1), Set(elts2)) = ...
    fun isMember(x, SET(elts)) = ...
end
```

- Types (compiler output)

```
type 'a set
val empty = - : 'a set
val insert = fn : 'a * ('a set) -> ('a set)
val union = fn : ('a set) * ('a set) -> ('a set)
val isMember = fn : 'a * ('a set) -> bool
```

Origin of Abstract Data Types

- Structured programming, data refinement
  - Write program assuming some desired operations
  - Later implement those operations
  - Example:
    - Write expression parser assuming a symbol table
    - Later implement symbol table data structure
- Research on extensible languages
  - What are essential properties of built-in types?
  - Try to provide equivalent user-defined types
  - Example:
    - ML sufficient to define list type that is same as built-in lists

Comparison with built-in types

- Example: int
  - Can declare variables of this type   x: int
  - Specific set of built-in operations   +, -, *, …
  - No other operations can be applied to integer values
- Similar properties desired for abstract types
  - Can declare variables x : abstract_type
  - Define a set of operations (give interface)
  - Language guarantees that only these operations can be applied to values of abstract_type

Modules

- General construct for information hiding
- Two parts
  - Interface:
    - A set of names and their types
  - Implementation:
    - Declaration for every entry in the interface
    - Additional declarations that are hidden

- Examples:
  - Modula modules, Ada packages, ML structures, ...

Modules and Data Abstraction

```
module Set
    interface
        type set
        val empty : set
        fun insert : elt * set -> set
        fun union : set * set -> set
        fun isMember : elt * set -> bool
    implementation
        type set = elt list
        val empty = nil
        fun insert(x, elts) = ...
        fun union(…) = ...
        ...
end Set
```

- Can define ADT
  - Private type
  - Public operations
- More general
  - Several related types and operations
- Some languages provide
  - Separate interface and implementation
  - One interface can have multiple implementations

Haskell modules

- Hide and selectively export declarations

Export list

```
module Tree ( Tree(Leaf,Branch), fringe ) where
    data Tree a   = Leaf a | Branch (Tree a) (Tree a)
    fringe :: Tree a -> [a]
    fringe (Leaf x)            = [x]
    fringe (Branch left right) = fringe left ++ fringe right
```

Declar-
ations

Basic description: http://www.haskell.org/tutorial/modules.html

More information: http://www.haskell.org/onlinereport/modules.html

Generic Abstractions

- Parameterize modules by types, other modules
- Create general implementations
  - Can be instantiated in many ways
- Language examples:
  - Ada generic packages, C++ templates, ML functors, …
  - ML geometry modules in supplementary readings
  - C++ Standard Template Library (STL) provides extensive examples

Summary

- Modularity
  - Interface, specification, and implementation
- Modular program development
  - Step-wise refinement ; Prototyping ; …
- Language support for modularity
  - Procedural abstraction
  - Abstract data types
    - Representation independence
    - Datatype induction
  - Packages and modules
  - Generic abstractions
    - Functions and modules with type parameters
- Modularity is supported by object-oriented languages, but did not originate with OOP