CS312

More Validation

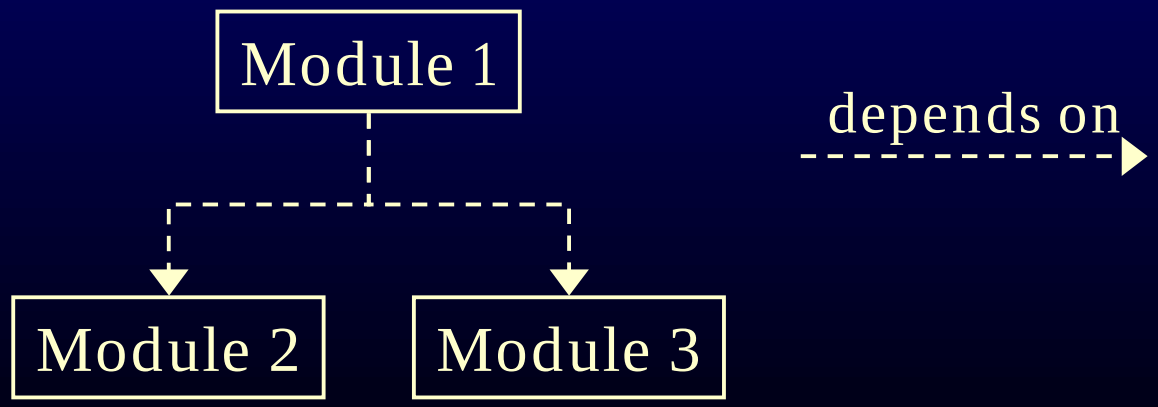
Lecture 11

24 Feb ’03

Modular structure

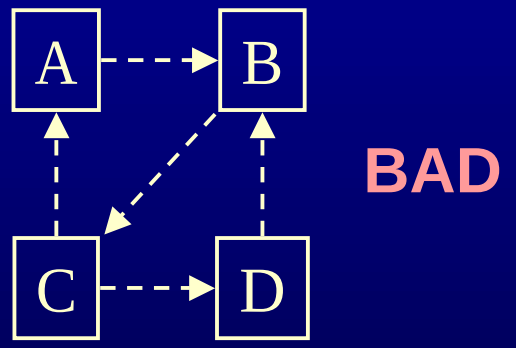
- Program is composed of modules
- One module *depends* on another if it uses a value, function, or type from it

- Module Dependency Diagram (MDD) helps understand large-scale program structure

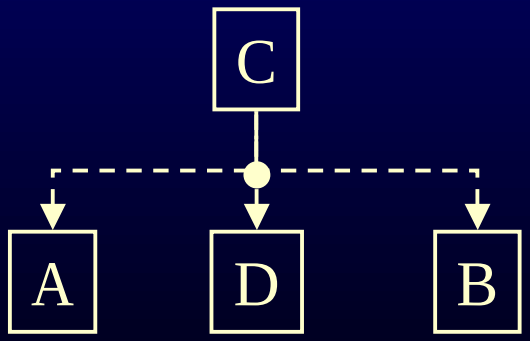


Keeping dependencies simple

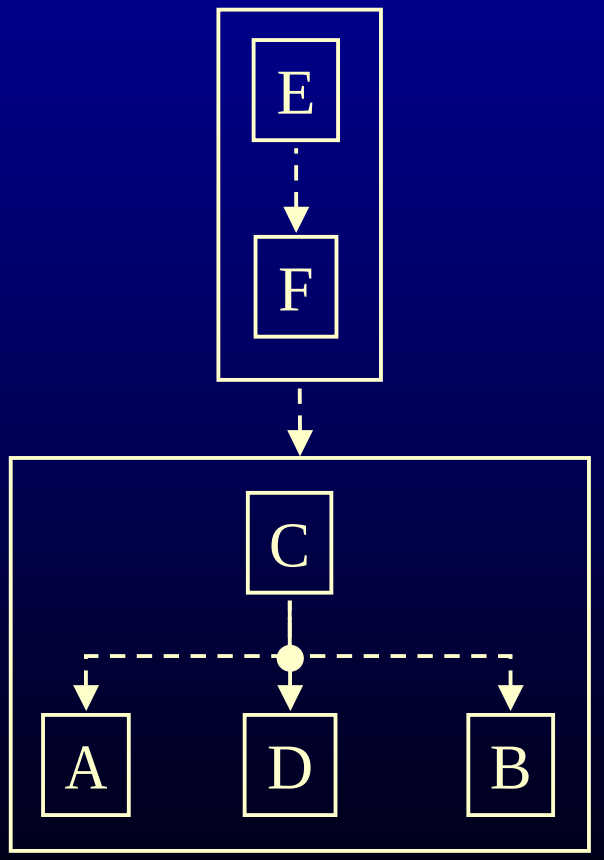
- Too many dependencies or cycles: harder to debug, maintain, extend software



BAD

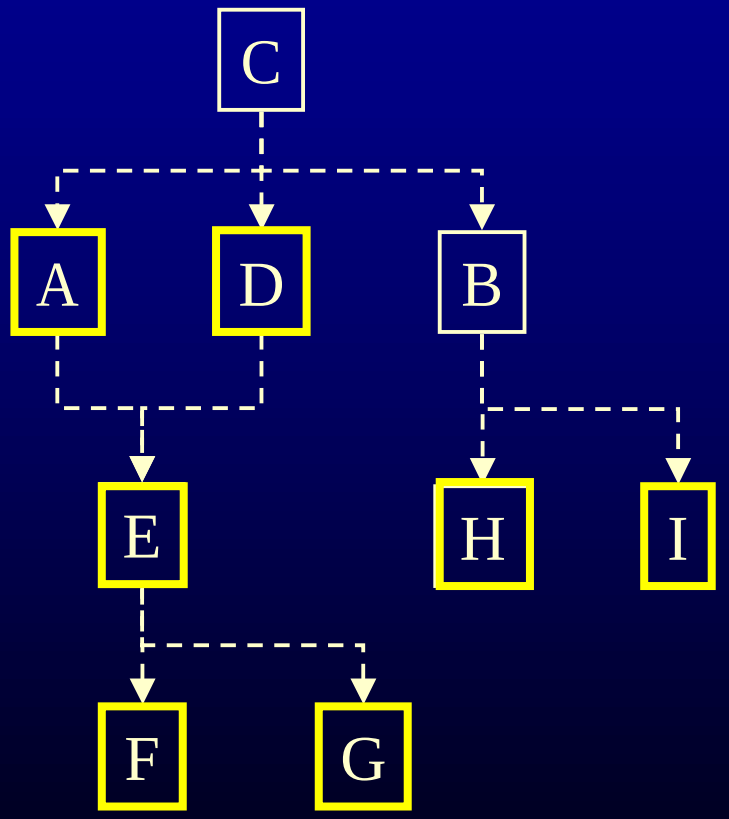


GOOD



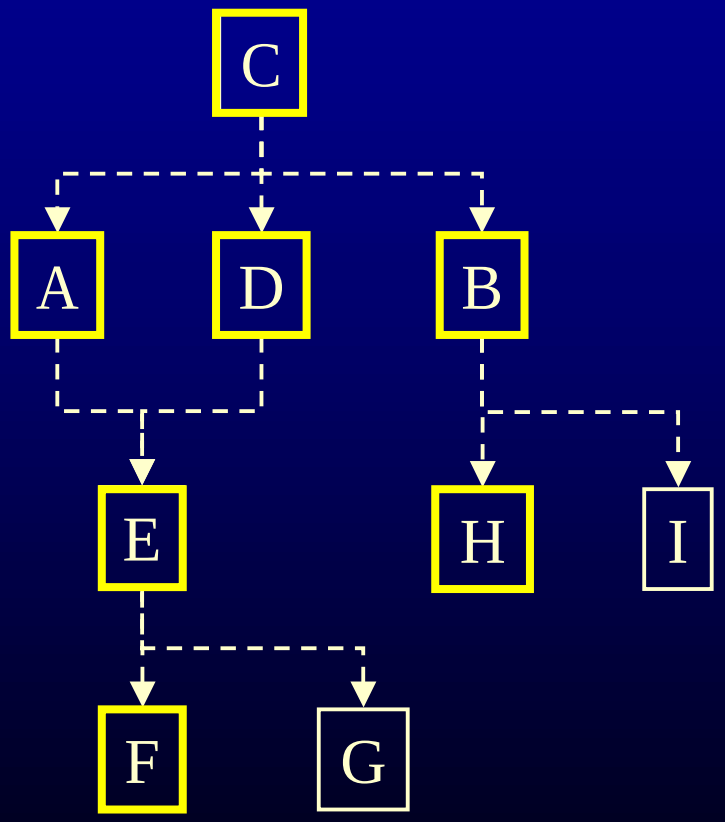
Bottom-up development

- Bottom-up: develop modules before the modules that depend on them
- Advantage: catch key technology/performance issues early
- Advantage: always working code, easy testing
- Disadvantage: catch large-scale design flaws late



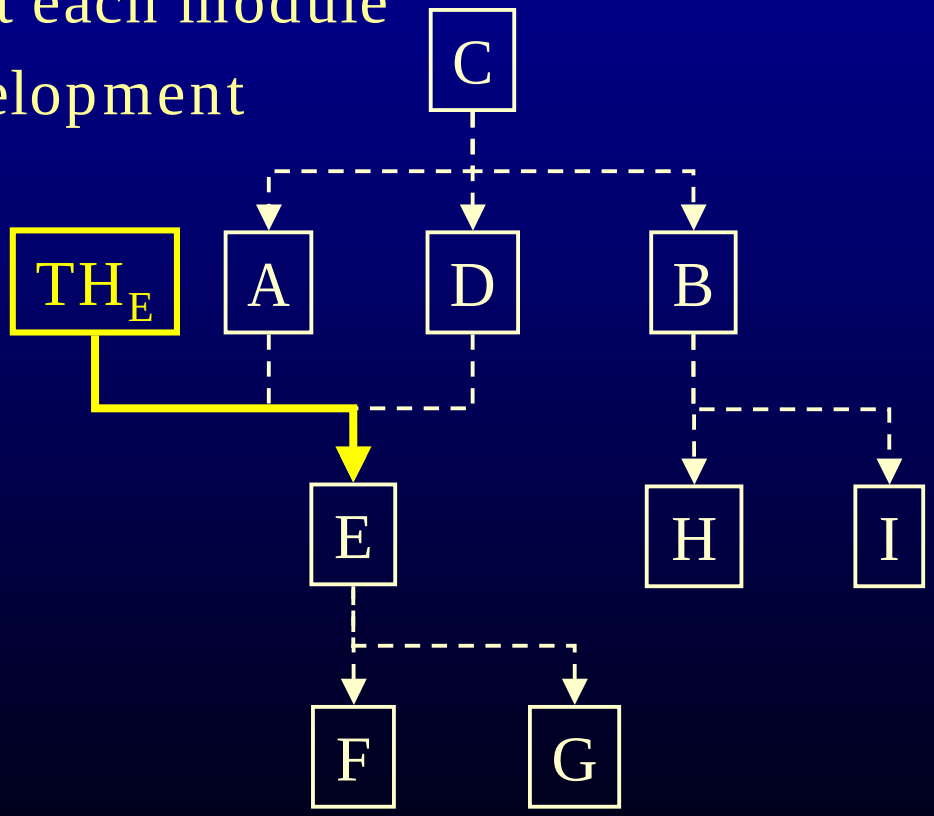
Top-down development

- Top-down: develop using modules before modules they depend on
- Advantage: get high-level design right from start, Advantage: easier to design interfaces well, quickly spec out system
- Disadvantage: harder to test until program complete



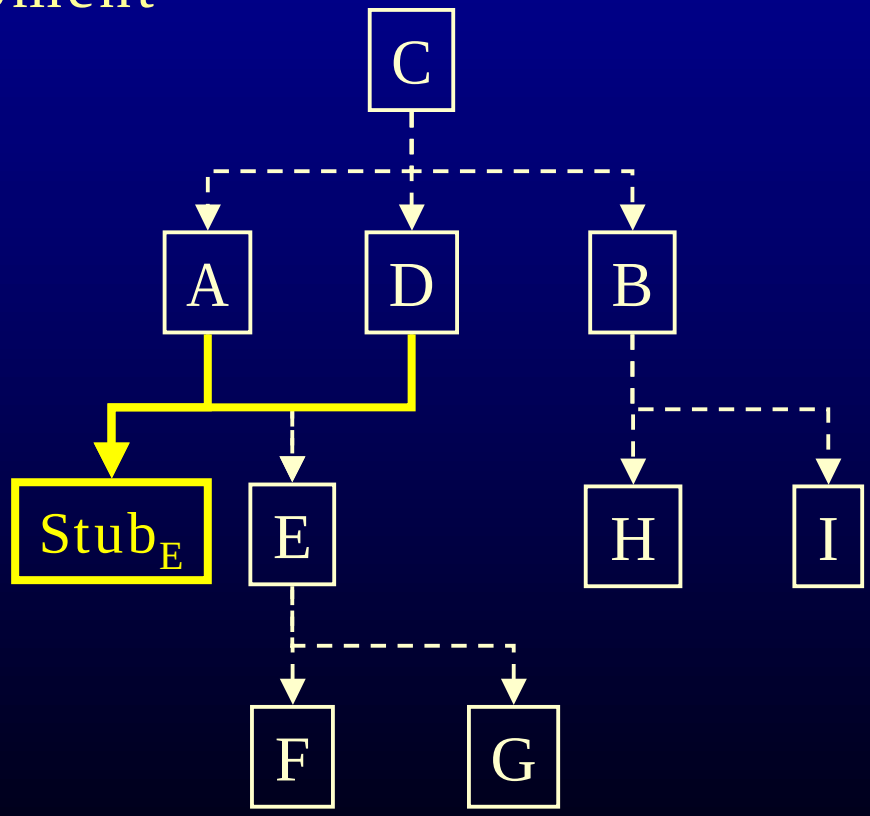
Unit testing

- Test modules through their interfaces
- Test each implementation against interface separately
- Write *test harness* to test each module
- Ideal for bottom-up development



Integration testing

- Test program from top level – validation of high-level structure, user interface of program
- Ideal for top-down development
- Replace missing module implementations with module *stubs* that simulate functionality to some extent



Top-down or bottom-up?

- Depends on the project!
  - Minimize risk: resolve uncertainties early
  - UI/high-level design: top-down
  - Core technology/performance: bottom-up
- Usually some mix of both strategies

  & both unit and integration testing

Verification

- Code verification gives extra confidence when testing is not enough
  - Maybe not possible to test adequately
  - Or code needs high assurance
- Goal: prove program works
  - Strategy: prove that each implementation satisfies its specification
  - Consider each module separately
    - Assume other modules satisfy *their* specifications
    - Works if no cycles in module dependency; otherwise may have to consider multiple modules at once

lmax example

- Does the following impl satisfy its spec?

```
(* lmax(lst) is the largest element
 * in lst. Requires: lst is non-nil.
 *)
fun lmax(lst: int list):int =
  case lst of
    [] => raise Fail "?"
  | [x] => x
  | h::t => Int.max(h, lmax(t))
```

Problem: Recursion leads to circular reasoning!

Proof by induction

Goal: prove some proposition is true for an infinite collection

- E.g., l**`max(lst)`** is the max element for *all* non-empty lists **`lst`**

1. State the proposition as a condition $P(n)$ that must be true for all $n \geq n_0$ (usually n_0 is 0 or 1)
   - n is the length of the list **`lst`** ($n \geq 1$)
   - $P(n)$ is: **`lmax(lst)`** is the max elem for all lists **`lst`** of length n
2. Base case: show $P(n_0)$
   - E.g., **`lmax(lst)`** is the max elem for all 1-elem lists **`lst`**
3. State *induction hypothesis* $P(n)$
   - Assume **`lmax(lst)`** is the max elem for all lists **`lst`** of length n
4. Induction step: show $P(n+1)$ assuming induction hypothesis
   - Show: **`lmax(lst)`** is the max elem for all $(n+1)$-elem lists **`lst`**
5. State conclusion: $P(n)$ is true for all $n \geq n_0$

$$P(1) \Rightarrow P(2) \Rightarrow P(3) \Rightarrow \ldots \Rightarrow P(n) \Rightarrow \ldots$$

"falling dominoes"

lmax

```
(* lmax(lst) is the largest element in lst.
 * Requires: lst is non-nil. *)
fun lmax(lst: int lst):int =
  case lst of
    [] => raise Fail "?"
  | [x] => x
  | h::t => Int.max(h, lmax(t))
```

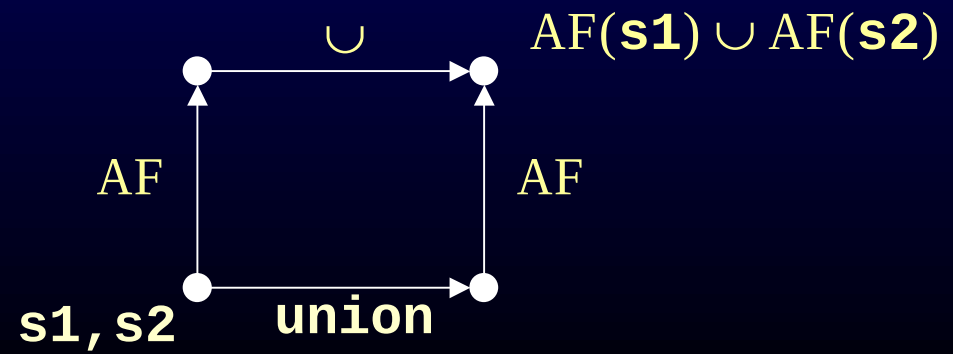
1. State the proposition: for all $n \geq 1$, **lmax(lst)** is the max elem for all lists **lst** of length n
2. Base case: is **lmax(lst)** give max elem for all 1-elem lists **lst?**
   - **lmax([**v**])** → **case [**v**] of …** → v
3. Induction hypothesis: **lmax(lst)** works for all **lst** of length n
4. Induction step: consider **lmax(lst)** where **lst** has length n+1
   - **lst** = **[**v_1,v_2,…v_{n+1}**]**
   - **lmax([**v_1,v_2,…v_{n+1}**])** → **case ([**v_1,v_2,…v_{n+1}**] of …** → **Int.max(**v_1**, lmax([**v_2,…v_{n+1}**]))**
   - IH: **lmax([**v_2,…v_{n+1}**]))** evaluates to maximum of v_2,…,v_{n+1}
   - If $v_1 \geq$ **lmax([**v_2,…v_{n+1}**])**, v_1 is max of v_1,…,v_{n+1}
5. Conclusion: **lmax** finds the max elem for all non-nil lists

Data abstraction

```
structure Natset = struct
  type set = int list
  (* AF: [x1,…,xn] represents {x1,…,xn} *)
  (* RI: no duplicates or negative elems *)
  fun union(s1: set, s2: set)=
    foldl (fn(x,s) => if contains(s,x) then s else x::s)
       s1 s2
```

`Natset.union` correct if:

Assuming RI(**`s1`**) & RI(**`s2`**), we can show:
RI(**`union(s1,s2)`**) &
AF(**`union(s1,s2)`**) = AF(**`s1`**) ∪ AF(**`s2`**)



Proof of correctness

- Given: $\mathbf{s_1}$ and $\mathbf{s_2}$ contain no negative elements or duplicates
- Show: RI(**union($\mathbf{s_1}$,$\mathbf{s_2}$)**) & AF(**union($\mathbf{s_1}$,$\mathbf{s_2}$)**) = AF($\mathbf{s_1}$) ∪ AF($\mathbf{s_2}$)
- **union($\mathbf{s_1}$,$\mathbf{s_2}$)** → **foldl (fn(x,s)=> if contains(s,x) then s else x::s) $\mathbf{s_1}$ $\mathbf{s_2}$**
- Now we need to use induction!

- State proposition in terms of P(n): for all $n \geq 0$, if RI($\mathbf{s_1}$) & RI($\mathbf{s_2}$) and $\mathbf{s_2}$ has length n, **foldl(…) $\mathbf{s_1}$ $\mathbf{s_2}$** evaluates to a list **l** such that RI(**l**) is true & AF(**l**) = AF($\mathbf{s_1}$) ∪ AF($\mathbf{s_2}$)
- Base case: **foldl(…) $\mathbf{s_1}$ [] evaluates to l=$\mathbf{s_1}$**
  RI($\mathbf{s_1}$) so RI(**l**), AF($\mathbf{s_1}$) ∪ AF(**[]**) = AF($\mathbf{s_1}$) ∪ ∅ = AF($\mathbf{s_1}$) = AF(**l**)
- Induction hypothesis: assume P(n)
- Induction step: assume RI($\mathbf{s_1}$) & RI($\mathbf{s_2}$) and $\mathbf{s_2}$=[v_1,…,v_{n+1}]
  - Recall: **foldl f b (h::t) → foldl f (f(h,b)) t**
  - **foldl (…) $\mathbf{s_1}$** [v_1,…,v_{n+1}]
    → **foldl (…) ((…)(v_1,$\mathbf{s_1}$))** [v_2,…,v_{n+1}]

Completing proof

```
fun union(s1: set, s2: set)=
  foldl (fn(x,s) => if contains(s,x) then s else x::s) s1 s2
```

- Given: $\mathbf{s_1}$ and $\mathbf{s_2}$ contain no negative elements or duplicates
- Show: RI(**union(**$\mathbf{s_1}$**,**$\mathbf{s_2}$**)**) & AF(**union(**$\mathbf{s_1}$**,**$\mathbf{s_2}$**)**) = AF($\mathbf{s_1}$) ∪ AF($\mathbf{s_2}$)

- Induction hypothesis P(n): if RI($\mathbf{s_1}$) & RI($\mathbf{s_2}$) and $\mathbf{s_2}$ has length n, **foldl(…)** $\mathbf{s_1}$ $\mathbf{s_2}$ evaluates to a list **l** such that RI(**l**) is true & AF(**l**) = AF($\mathbf{s_1}$) ∪ AF($\mathbf{s_2}$)
- Induction step, show P(n+1): assume RI($\mathbf{s_1}$) & RI($\mathbf{s_2}$) and $\mathbf{s_2}$=**[**v_1**,**…**,**v_{n+1}**]**
  - **foldl (…)** $\mathbf{s_1}$ **[**v_1**,**…**,**v_{n+1}**]**
    → **foldl (…) ((…)(**v_1**,** $\mathbf{s_1}$**))** **[**v_2**,**…**,**v_{n+1}**]**
    → **foldl (…) (if contains(**$\mathbf{s_1}$**,**v_1**) then** $\mathbf{s_1}$ **else** v_1**::**$\mathbf{s_1}$**))**
    [v_2**,**…**,**v_{n+1}**]**
  - Have RI($\mathbf{s_1}$) , so we can assume **contains** works
    if contains($\mathbf{s_1}$**,**v_1**) then** $\mathbf{s_1}$ **else** v_1**::**$\mathbf{s_1}$ → $\mathbf{s_1}'$ where RI($\mathbf{s_1}'$) and AF($\mathbf{s_1}'$) = AF($\mathbf{s_1}$) ∪ {v_1}
  - Now, can use induction hypothesis on **foldl (…)** $\mathbf{s_1}'$**[**v_2**,**…**,**v_{n+1}**]** – it evaluates to a list **l** such that
    RI(**l**) & AF(**l**) = AF($\mathbf{s_1}'$) ∪ AF(**[**v_2**,**…**,**v_{n+1}**]**) = AF($\mathbf{s_1}$) ∪ {v_1} ∪{ v_2**,**…**,**v_{n+1}}
    = AF($\mathbf{s_1}$) ∪ AF($\mathbf{s_2}$)
  - This **l** is the result of **union(s1,s2)** – we're done!

Some thoughts

- We can really prove code works!
- Convincing proof requires knowing evaluation rules for language
- Almost any interesting code requires proof by induction
- Using recursive functions, loops correctly requires inductive reasoning – you have already (partly) internalized this process
- Reasoning explicitly avoids errors