Why Modularity?

• Aimed at various needs
  – Namespace management
  – Program organization
  – Information hiding and data abstraction
  – Separate compilation
Namespace Management

- In C, all functions are at top level and (by default) globally accessible
  - Potential for name clashes
    - Between files by different programmers
    - Between user code and libraries
  - Requires care to avoid collisions
    - Special variable name conventions
      _exit
      XtSetValues XtCreateManagedWidget
    - Declaring functions static when possible

Encapsulation

- Information Hiding
  - Concealing internal implementation details (like data representations or helper functions)
  - Have access to more information when inside a module than when outside
- Key idea: interfaces
  - What is visible to the outside world?
  - What can a user depend upon?
  - Where is this enforced?
Interfaces

• Access restrictions part of module definition
  - Java classes: public, private, protected, default
• Or, separate interfaces
  - C++: Namespace specification
    namespace Stack {
    struct Rep;
    typedef Rep& stack;
    void push(stack s, char c);
    ...
    }
  - SML: Signatures

Package-like Constructs

• C++
  - Namespaces
  - Classes
• Java
  - Packages
  - Classes
• SML
  - Structures
• Modula-3: Modules, Perl: Packages, etc.
Open and Closed Definitions

• Can package definitions be broken up?

• Closed packages: definition all in one place
  - e.g., C++ classes and SML structures

• Open packages: packages are extensible
  - Can be split up among separately-compiled files
  - e.g., C++ namespaces and Java packages

SML Specifics
Structures

- A structure is a collection of definitions
  - Structures are not ordinary values like lists or
tuples that can be passed around at run-time.
  - Hence not the analogue of `struct` in C/C++

```plaintext
define IntQueue =
struct
  type queue = int list
  val empty : queue = []
  fun dequeue(h::t) = (h,t)
  fun enqueue(q, x) = q @ [x]
end
```

Structure Definitions

We can give a name to a structure.

```plaintext
define IntQueue =
structure
  type queue = int list
  val empty : queue = []
  fun dequeue(h::t) = (h,t)
  fun enqueue(q, x) = q @ [x]
end
```

Now we can refer to the type `IntQueue.queue` (which is
interchangeable with `int list`), to the function
`IntQueue.enqueue`, and so on.
Standard Basis Library

- Many structures have been predefined for you:
  - Int
  - Real
  - String
  - List
  - TextIO
- So you can freely use functions like
  - `Int.toString : string -> int`
  - `List.length : 'a list -> int`
- See relevant web pages for more details

Specifications and Signatures

- A specification is a description of a structure component.
- Intuitive examples:
  - "x has type int"
  - "queue is a type"
  - "queue is synonymous with the type int list"
  - "option is a polymorphic datatype with constructors NONE and SOME"
Signatures

A signature is a collection of specifications

```plaintext
sig
    type queue
    val empty   : queue
    val dequeue : queue -> int * queue
    val enqueue : queue * int -> queue
end
```

(The `fun` keyword never appears in a specification; just `val`.)

Signature Definitions

We can also give a name to a signature.

```plaintext
signature INTQUEUE =
    sig
        type queue
        val empty   : queue
        val dequeue : queue -> int * queue
        val enqueue : queue * int -> queue
    end
```
Satisfaction

- We say that a structure $M$ satisfies a signature $SIG$ if the structure contains (at least) components matching all the specifications in the given signature.
  - The structure $M$ may contain other components too.

- In SML, many signatures may satisfy the same structure
  - Different views of the same structure, providing different amounts of detail.

Information Hiding

- Suppose $M$ satisfies $SIG$.
- Then $M :> SIG$ is also a structure
  - The result of ascribing the signature $SIG$ to $M$.
  - New structure exposes only the information given by $SIG$.

- Why would we do this?
  - The rest of the program can only depend on the information in $SIG$
  - Ensures that the internal implementation of $M$ can change without breaking code that uses the module.
Examples

• Assume we have the definition

```plaintext
structure IQueue =
  struct
    type queue = int list
    val empty : queue = []
    fun dequeue(h::t) = (h,t)
    fun enqueue(x, q) = q @ [x]
  end
```

Example 1

```plaintext
signature SIG1 =
sig
  type queue = int list
  val empty   : int list
  val dequeue : int list -> int * int list
  val enqueue : int list * int -> int list
end
structure S1 = IQueue :> SIG1
```

This signature ascription doesn't hide anything. For example, S1.queue = IQueue.queue = int list and S1.empty : int list
Example 2

signature SIG2 =
sig
  type queue = int list
  val empty : queue
  val dequeue : queue -> int * queue
  val enqueue : queue * int -> int list
end
structure S2 = IQueue :> SIG2

This signature ascription doesn't hide anything either, since SIG1 and SIG2 are the same signature.

Example 3

signature SIG3 =
sig
  type queue = int list
  val empty : queue
  val dequeue : queue -> int * queue
end
structure S3 = IQueue :> SIG3

Hmm...now there is still a function IQueue.enqueue but we cannot refer to S3.dequeue because it's been hidden.
Example 4

signature SIG4 =
sig
type queue
val empty : queue
val dequeue : queue -> int * queue
val enqueue : queue * int -> int list
end
structure S4 = IQueue => SIG4

Aha...now we've actually done something useful! The type S4.queue is now abstract; we've "hidden" the fact that S4 implements queues with lists.
IQueue.dequeue [3,4,5] typechecks.
S4.dequeue [3,4,5] does not

Example 5

signature SIG5 =
sig
type queue
val dequeue : queue -> int * queue
val enqueue : queue * int -> int list
end
structure S5 = IQueue => SIG5

S5 is completely useless. Why?
Alternate Implementation

```plaintext
structure IntQueue' =
  struct
    (* Invariant: the pair
       ([x1,...,xn],[y1,...,yn])
       represents (from front to back)
       x1, ..., xn, yn, ..., y1 *)
    type queue = int list * int list
    val empty : queue = ([],[])
    fun enqueue(x, (f,b)) = (f,x::b)
    fun dequeue(f::fs,b) = (f,(fs,b))
    | dequeue([],b) = dequeue(rev b,[])
  end :> SIG4
```

A Dictionary Signature

- Also known as lookup tables or environments
  - Associates values with keys (strings)

```plaintext
sig
  type 'a dict
  val empty : 'a dict
  val insert : 'a dict * string * 'a -> 'a dict
  val lookup : 'a dict * string -> 'a
end
```
Alternative Signature

• Could emphasize keys are strings:
  – Places more requirements upon an implementation.

```ocaml
signature STRINGDICT = sig
  type key = string
  type 'a dict
  val empty : 'a dict
  val insert : 'a dict * key * 'a -> 'a dict
  val lookup : 'a dict * key -> 'a
end
```

Alternative Interface

• Now only need to change one line to specify that keys are integers.

```ocaml
signature INTDICT = sig
  type key = int
  type 'a dict
  val empty : 'a dict
  val insert : 'a dict * key * 'a -> 'a dict
  val lookup : 'a dict * key -> 'a
end
```
Generic Dictionary Interface

- Any implementation satisfying `STRINGDICT` or `INTDICT` also satisfies the following less-precise signature:

```ocaml
signature DICT = sig
  type key
  type 'a dict
  val empty : 'a dict
  val insert : 'a dict * key * 'a -> 'a dict
  val lookup : 'a dict * key -> 'a
end
```

Signature Patching

- A program might use many different dictionaries with keys of different types.
- Painful/error-prone to re-type all the functions for each signature.
- However, we can define `INTDICT` and `STRINGDICT` as specializations of the `DICT` signature.
Signature Patching

```
signature DICT = sig
  type key
type 'a dict
val empty : 'a dict
val insert : 'a dict * key * 'a -> 'a dict
val lookup : 'a dict * key -> 'a
end

signature INTDICT =
  DICT where type key = int

signature STRINGDICT =
  DICT where type key = string
```

Functors

- Parameterized structures
  ```
  functor FunctorName(specifications) = structure
  ```

- Why?
  - For example, definitions of structures satisfying
    INTDICT and STRINGDICT will share most of the
    same code
    - Just replace integer equality with string equality
  - Idea: a dictionary structure generator
    - Given information about keys, create dictionary module
A Dictionary Functor

functor Dict\(\text{type } t\)
  \(\text{val } \text{eq} : t * t \rightarrow \text{bool}\) =
  \text{struct}
  \(\text{type } \text{key} = t\)
  \(\text{type } 'a \text{ dict} = (\text{key} * 'a) \text{ list}\)
  \(\text{val } \text{empty} = []\)
  \(\text{fun } \text{insert}(d:'a \text{ dict}, k:t, x:'a) = (k,x)::d\)
  \(\text{fun } \text{lookup}'((k,x)::\text{rest},k') =\)
  \(\text{if } (\text{eq}(k,k')) \text{ then } \text{else } \text{lookup}'(\text{rest},k')\)
  \(\text{fun } \text{lookup}(d,k) = \text{lookup}'(d,k)\)
  \text{end}

Applying a Functor

\text{structure } \text{StringDict} =
\text{Dict(\text{type } t = \text{string}}
\text{fun } \text{eq}(s1:\text{string},s2:\text{string}) = (s1=s2)\)
\text{structure } \text{IntDict} =
\text{Dict(\text{type } t = \text{int}}
\text{fun } \text{eq}(n1:t,n2:t) = (n1=n2)\)
Applying a Functor

```ml
structure StringDict =
  Dict(type t = string
       fun eq(s1:string,s2:string) = (s1=s2))
structure IntDict =
  Dict(type t = int
       fun eq(n1:t,n2:t) = (n1=n2))
```

The Dict functor as defined probably exposes too much information. For example

'\texttt{a Stringdict.dict} = (\texttt{string} * 'a) \texttt{list}
and we can call \texttt{StringDict.lookup}'

---

Improving the Definition

```ml
signature DICT = sig
  type key
  type 'a dict
  val empty  : 'a dict
  val insert : 'a dict * key * 'a -> 'a dict
  val lookup : 'a dict * key -> 'a
end

functor Dict(type t
              val eq : t * t -> bool) =
  struct
    type key = t
    type 'a dict = (key * 'a) \texttt{list}
    val empty = []
    ...etc...
  end :> DICT (* Does this work?? *)
```
Improving the Definition

```ml
functor Dict (type t
  val eq : t * t -> bool) =
struct
  type key = t
  type 'a dict = (key * 'a) list
  val empty = []
  ...etc...
end ?: DICT where type key = t
```

Improving the Definition

```ml
structure StringDict =
  Dict (type t = string
    fun eq(s1:string,s2:string) = (s1=s2))
structure IntDict =
  Dict (type t = int
    fun eq(n1:t,n2:t) = (n1=n2))
```

Now
  StringDict.key = string and IntDict.key = int
but
  StringDict.dict is abstract
  IntDict.dict is abstract
  IntDict.lookup' is not accessible