Modularity

• A modularity mechanism is, most generally, a way to collect together related pieces of code.

• What functions do these serve?
**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

**Abstraction**

- 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, package
- Or, separate interfaces
  - C++: Namespace specification
    ```c++
    namespace Stack {
      struct Rep;
      typedef Rep& stack;
      void push(stack s, char c);
      ...
    }
    ```
  - SML: Signatures

**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
Structures

• A structure is a collection of definitions
  - Structures are not ordinary values
  - Also not the analogue of struct in C/C++
    • SML does have records, which are analogous

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

Using Structures

• References to components can be painfully long
  ```sml```
  Posix.FileSys.O.append
  ```sml```
• Three solutions:
  - Define new variables
    ```sml```
    val append = Posix.FileSys.O.append
    ```sml```
  - Give structures short names:
    ```sml```
    structure F = Posix.FileSys
    structure O = Posix.FileSys.O
    ```sml```
    • Then we can refer to F.O.append or O.append
  - Use open
    ```sml```
    open Posix.FileSys.O
    ```sml```
    • Now append refers to Posix.FileSys.O.append, but there are others as well: sync now refers to Posix.FileSys.O.sync
    • Dangerous, from a software-engineering perspective.

Structure Definitions

We can give a name to a structure.

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

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 are predefined for you:
  ```sml```
  Int
  Real
  String
  List
  TextIO
  ```sml```
• So you can freely use functions like
  ```sml```
  Int.toString : int -> string
  List.length : 'a list -> int
  ```sml```
• See web pages for more details
  - Last link on the main 131 page.
Specifications and Signatures

- A specification is a description of a structure component.

- Intuitive examples:
  "x has type int"
  "queue is a type"
  "queue is a type synonymous with the type int list"
  "option is a polymorphic datatype with constructors NONE and SOME"

Signatures

A signature is a collection of specifications

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

(Signatures use `val` for definitions, not `fun`.)

Signature Definitions

We can also give a name to a signature.

```sml
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 \( \text{SIG} \) if the structure contains components matching all the specifications in the given signature.
  - The structure \( M \) may contain other definitions as well.
  - The structure \( M \) does not have to know about this signature

- Signature satisfaction is a "structural" property.
  - In SML, many structures may satisfy the same signature
  - The same structure satisfies many different signatures
    - Different views of the same structure, providing different amounts of detail.
Information Hiding

- Suppose \( M \) satisfies \( \text{SIG} \).
- Then \( M :> \text{SIG} \) is also a structure
  - The result of ascribing the signature \( \text{SIG} \) to \( M \).
  - New structure exposes only the information given by \( \text{SIG} \).
- Why would we do this?
  - The rest of the program can only depend on the information in \( \text{SIG} \).
  - Ensures that the internal implementation of \( M \) can change without breaking code that uses the module.

Examples

- Assume we have the definition

```haskell
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

```haskell
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, \( S1.queue = IQueue.queue = \text{int list} \) and \( S1.empty : \text{int list} \).

Example 2

```haskell
signature SIG2 =
sig
    type queue = int list
    val empty   : queue
    val dequeue : queue -> int * queue
    val enqueue : queue * int -> queue
end

structure S2 = IQueue :> SIG2
```

This signature ascription doesn't hide anything either, since \( \text{SIG1} \) and \( \text{SIG2} \) are the same signature.
Example 3

```ml
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 $S3$.enqueue because it's been hidden. (We can still cons onto the front of a queue though.)

Example 4

```ml
signature SIG4 =
  sig
    type queue
    val empty : queue
    val dequeue : queue -> int * queue
    val enqueue : queue * int -> queue
  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

```ml
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

```ml
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 = ([],[])
    val dequeue : queue -> int * queue
      val enqueue : queue * int -> int list
  end :> SIG4
```

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)

```
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.

```
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.

```
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

```
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.

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(type t

  val eq : t * t -> bool) =

  struct
    type key = t
    type 'a dict = (key * 'a) list
    val empty = []
    fun insert(d:'a dict, k:t, x:'a) = (k,x)::d
    fun lookup'((k,x)::rest,k') =
      if (eq(k,k')) then x else lookup'(rest,k')
    fun lookup(d,k) = lookup'(d,k)
  end
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))
```

FunctorName (definitions)

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) list
    val empty = []
    ...
  end
end :> DICT
```

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

```ml
int Stringdict.dict = (string * int) list
and we can call the helper function StringDict.lookup'
```

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 = []
    ...
  end
  :> DICT
```

where type key = t
Alternate Syntax

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

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 (and different)
IntDict.lookup' is not accessible