Computer Science 131
Programming Languages

October 19, 2000
Modularity
Part 1: Generalities
Modularity Mechanisms

- Wide Variety
  - Files
  - Modules
  - Packages
  - Namespaces
  - Classes

- Fulfill different needs to different degrees
  - Namespace management
  - Program organization
  - Information hiding and data abstraction
Namespace Management

• In C, all functions are at top level and (by default) globally accessible
  - Large 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
Solution: Packages

• (Named) aggregate of values and functions
  - Possibly may contain types
  - Usually can be nested
• Need only worry about clashes between top level packages.
  - All other names are qualified
    
      Java.lang.String

      std::printf

• Usually can limit external access to components
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
  - One file (modulo `#include`) specifies all components
  - 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
Encapsulation

• Information Hiding
  - Hiding internal implementation details
  - Data representations, helper functions

• Modules restrict scope of definitions
  - Have access to more information when inside module than when outside

• Key idea: interfaces
  - What is visible to the outside world?
  - What can a user depend upon?
  - Who enforces this?
Interfaces

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

• Both
  - Serve as collections of code
  - Can provide information hiding

• But classes
  - Are organized around operations on one type
  - Usually aimed at generating multiple objects
  - Allow inheritance/overriding (anti-modular?)

• Packages
  - Can contain code for multiple types
  - Generally one instance of the package
Part 2: Modules in SML
Structures

• A structure is a collection of definitions

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

• We can give a name to a structure

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

• Then refer to `IntQueue.empty` and so on
Specifications

• A specification is a description of a binding.
• For example, one specification for
  
  ```
  val x = 3
  ```

  would be

  ```
  val x : int
  ```

• And a specification for
  
  ```
  val x = "hello " ^ "world"
  ```

  would be

  ```
  val x : string
  ```
Specifications

• A specification for the definition
  \[
  \text{fun succ}(n) = n+1
  \]
  would be
  \[
  \text{val succ : int -> int}
  \]
• Note the \texttt{val} keyword; \texttt{x} is a variable that is bound to a function value
Specifications

- Possible type specifications for the definition:
  
  \[
  \text{type intpair} = \text{int} \times \text{int}
  \]

  would be

  \[
  \begin{align*}
  \text{type intpair} \\
  \text{type intpair} = \text{int} \times \text{int}
  \end{align*}
  \]

- The latter specification is more informative
Specifications

• Possible type specifications for the definition

datatype 'a option = NONE | SOME of 'a

would be

    type 'a option

datatype 'a option = NONE | SOME of 'a

• Or even

    type 'a option
    val NONE : 'a option
    val SOME : 'a -> 'a option
Signatures

• A signature is the interface for a structure
  - Defined by specifications for components
• For example, `IntQueue` satisfies the signature

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

• **IntQueue** also satisfies the signature

```ocaml
sig
  type queue = int list
  val empty : queue
  val dequeue : queue -> int * queue
  val enqueue : int * queue -> queue
end
```
Information Hiding

- A structure satisfies a signature iff it contains everything required in the signature.
- Signatures can contain less information than is in the structure.
  - Signature can omit components
  - Signature can omit definitions of types
Information Hiding

• For example, \texttt{IntQueue} also satisfies:

\begin{verbatim}
  sig
    type queue = int list
    val dequeue : queue -> int * queue
    val enqueue : int * queue -> queue
  end
\end{verbatim}
Information Hiding

...as well as the signature:

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

...and the rather unfortunate signature:

```ocaml
sig
  type queue
  val dequeue : queue -> int * queue
  val enqueue : int * queue -> queue
end
```
Signature Definitions

• Analogous to the definition
  
  \[
  \text{structure IntQueue = struct ... end}
  \]

• We can give signatures a name:
  
  \[
  \text{signature INTQUEUE =}
  \begin{align*}
  &\text{sig} \\
  &\text{type queue} \\
  &\text{val empty : queue} \\
  &\text{val dequeue : queue \rightarrow int * queue} \\
  &\text{val enqueue : int * queue \rightarrow queue}
  \end{align*}
  \]
Signature Ascription

• After the definition

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

the type IntQueue.queue and int list are interchangeable
```
Signature Ascription

• But the definition

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

hides all information except that specified in the signature INTQUEUE.
```
Example: Dictionaries

• A module that implements environments as we've seen in the past might have signature

```ml
sig
  type 'a dict
  val empty : 'a dict
  val insert : 'a dict * string * 'a -> 'a dict
  val lookup : 'a dict * string -> 'a
end
```
Example: Dictionaries

• A slightly different version emphasizes this is a lookup table where the keys are strings:

```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
```
Example: Dictionaries

- A small change would give the interface for a lookup table where 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
```
Example: Dictionaries

- An implementation satisfying `STRINGDICT` or `INTDICT` satisfies the 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