Review

Last week

• ML

Outline

• Modules
• Next Programming Assignment

Sources:
PLP, 3.3.4, 3.3.5, 3.8

http://en.wikipedia.org/wiki/Argument_dependent_name_lookup

Software Complexity

• Tony Hoare:
  There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other is to make it so complicated that there are no obvious deficiencies.

• Edsger Dijkstra:
  Computing is the only profession in which a single mind is obliged to span the distance from a bit to a few hundred megabytes, a ratio of $1$ to $10^9$, or nine orders of magnitude. Compared to that number of semantic levels, the average mathematical theory is almost flat. By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history.

• Steve McConnell:
  Software’s Primary Technical Imperative has to be managing complexity.
Dealing with Complexity

Problem Decomposition: Minimize the amount of essential complexity that has to be dealt with at any one time. In most cases, this is the top priority.

Information Hiding: Encapsulate complexity so that it is not accessible outside of a small part of the program.

Additional benefits of information hiding:
- Reduces risk of name conflicts
- Safeguards integrity of data
- Helps to compartmentalize run-time errors

What is a module?
- different languages use different terms
- different languages have different semantics for this construct (sometimes very different)
- a module is somewhat like a record, but with an important distinction:
  - record ➞ consists of a set of names called fields, which refer to values in the record
  - module ➞ consists of a set of names, which can refer to values, types, routines, other language-specific entities, and possibly other modules

Modules

A module is a programming language construct that enables problem decomposition and information hiding.

A module:
- defines a set of logically related entities (strong internal coupling)
- has a public interface that defines entities exported by the component
- may depend on the entities defined in the interface of another component (weak external coupling)
- may include other (private) entities that are not exported (information hiding)

Language constructs for modularity

Issues:
- public interface
- private implementation
- dependencies between modules
- naming conventions of imported entities
- relationship between modules and files

access control: module controls whether a client can access its contents
- closed module: names must be explicitly imported from outside module
- open module: outside names are accessible inside module
Language choices

- **ADA**: package declaration and body, *with* and *use* clauses, renamings
- **C**: header files, `#include` directives
- **C++**: header files, `#include` directives, namespaces, *using* declarations/directives, namespace alias definitions
- **JAVA**: packages, *import* statements
- **ML**: *signature*, *structure* and *functor* definitions

### ADA: Packages

```ada
package Queues is
    Size: constant Integer := 1000;

type Queue is private; -- information hiding

procedure Enqueue (Q: in out Queue, Elem: Integer);
procedure Dequeue (Q: in out Queue; Elem: out Integer);
function Empty (Q: Queue) return Boolean;
function Full (Q: Queue) return Boolean;
function Slack (Q: Queue) return Integer;
    -- overloaded operator "=":
function "=" (Q1, Q2: Queue) return Boolean;

private
    ... -- concern of implementation, not of package client
end Queues;
```

### Private parts and information hiding

```ada
package Queues is
    ... -- visible declarations

private
    type Storage is
        array (Integer range <>) of Integer;

type Queue is record
    Front: Integer := 0; -- next elem to remove
    Back: Integer := 0; -- next available slot
    Contents: Storage (0 .. Size-1); -- actual contents
    Num: Integer := 0;
end record;
end Queues;
```

### Implementation of Queues

```ada
package body Queues is

procedure Enqueue (Q: in out Queue; Elem: Integer) is
    begin
        if Full(Q) then
            -- need to signal error: raise exception
        else
            Q.Contents(Q.Back) := Elem;
            Q.Back := (Q.Back + 1) mod Size;
        end if;
    end Enqueue;
```
Predicates on queues

**function** Empty (Q: Queue) **return** Boolean **is**
begin
  return Q.Num = 0;  -- client cannot access Num directly
end Empty;

**function** Full (Q: Queue) **return** Boolean **is**
begin
  return Q.Num = Size;
end Full;

**function** Slack (Q: Queue) **return** Integer **is**
begin
  return Size - Q.Num;
end Slack;

Operator Overloading

**function** "=" (Q1, Q2 : Queue) **return** Boolean **is**
begin
  if Q1.Num /= Q2.Num then
    return False;
  else
    for J in 1 .. Q1.Num loop
      -- check corresponding elements
      if Q1.Contents((Q1.Front + J - 1) mod Size) /=
         Q2.Contents((Q2.Front + J - 1) mod Size)
        then
          return False;
        end if;
    end loop;
    return True;  -- all elements are equal
  end if;
end "=";  -- operator "/=" implicitly defined
           -- as negation of "="

Client can only use visible interface

**with** Queues;  **use** Queues;  **with** Text_IO;

**procedure** Test **is**
Q1, Q2: Queue;  -- local objects of a private type
Val: Integer;
begin
  Enqueue(Q1, 200);  -- visible operation
  for J in 1 .. 25 loop
    Enqueue(Q1, J);
    Enqueue(Q2, J);
  end loop;
  Dequeue(Q1, Val);  -- visible operation
  if Q1 /= Q2 then
    Text_IO.Put_Line("lousy\_implementation");
  end if;
end Test;

Implementation

- package body holds bodies of subprograms that implement interface
- package may not require a body:

  **package** Days **is**
  type Day **is** (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  subtype Weekday **is** Day **range** Mon .. Fri;
  Tomorrow: **constant** array (Day) **of** Day
            := (Tue, Wed, Thu, Fri, Sat, Sun, Mon);
  Next\_Work\_Day: **constant** array (Weekday) **of** Weekday
                 := (Tue, Wed, Thu, Fri, Mon);
end Days;
### Syntactic sugar: use and renames

Visible entities can be denoted with an expanded name:

```pascal
with Text_IO;
...
Text_IO.Put_Line("hello");
```

**use** clause makes name of entity directly usable:

```pascal
with Text_IO; use Text_IO;
...
Put_Line("hello");
```

**renames** clause makes name of entity more manageable:

```pascal
with Text_IO;
package T renames Text_IO;
...
T.Put_Line("hello");
```

### Sugar can be indispensable

```pascal
with Queues;

procedure Test is
Q1, Q2: Queues.Queue;
begin
if Q1 = Q2 then ...
  -- error: "=" is not directly visible
  -- must write instead: Queues."="(Q1, Q2)
```

Two solutions:

- import all entities:
  ```pascal
  use Queues;
  ```

- import operators only:
  ```pascal
  use type Queues.Queue;
  ```

### C++ namespaces

- late addition to the language
- an entity requires one or more declarations and a single definition
- a namespace declaration can contain both, but definitions may also be given separately

```
// in .h file
namespace util {
  int f (int); /* declaration of f */
}

// in .cpp file
namespace util {
  int f (int i) {
    // definition provides body of function
    ...
  }
```

### Dependencies between modules in C++

- files have semantic significance: **#include** directives means textual substitution of one file in another
- convention is to use header files for shared interfaces

```pascal
#include <iostream> // import declarations

int main () {
  std::cout << "C++ is really different"
  << std::endl;
}
```
Header files are visible interfaces

```cpp
namespace stack { // in file stack.h
    void push (char);
    char pop ();
}
```

```cpp
#include "stack.h" // import into client file

void f () {
    stack::push('c');
    if (stack::pop() != 'c') error("impossible");
}
```

Namespace Definitions

```cpp
#include "stack.h" // import declarations
namespace stack { // the definition
    const unsigned int MaxSize = 200;
    char v[MaxSize];
    unsigned int numElems = 0;
    void push (char c) {
        if (numElems >= MaxSize)
            throw std::out_of_range("stack overflow");
        v[numElems++] = c;
    }
    char pop () {
        if (numElems == 0)
            throw std::out_of_range("stack underflow");
        return v[--numElems];
    }
}
```

Syntactic sugar: using declarations

```cpp
namespace queue { // works on single queue
    void enqueue (int);
    int dequeue ();
}
```

```cpp
#include "queue.h" // in client file
using queue::dequeue; // selective: a single entity

void f () {
    queue::enqueue(10); // prefix needed for enqueue
    queue::enqueue(-999);
    if (queue::dequeue() != 10) // but not for dequeue
        error("buggy implementation");
}
```

Wholesale import: the using directive

```cpp
#include "queue.h" // in client file

using namespace queue; // import everything

void f () {
    enqueue(10); // prefix not needed
    enqueue(-999);
    if (dequeue() != 10) // for anything
        error("buggy implementation");
}
```
Shortening names

Sometimes, we want to qualify names, but with a shorter name.

In Ada:

   package PN renames A.Very_Long.Package_Name;

In C++:

   namespace pn = a::very_long::package_name;

We can now use PN as the qualifier instead of the long name.

Visibility: Koenig lookup in C++

When an unqualified name is used as a function call, other namespaces besides those currently being used may be searched; this search depends on the types of the arguments to the function.

This is known as Koenig lookup or argument dependent name lookup.

For each argument type \( T \) in the function call, there is a set of zero or more associated namespaces to be considered. The set of namespaces is determined entirely by the types of the function arguments.

Type-def names used to specify the types do not contribute to this set.

Koenig lookup: details

The set of namespaces are determined in the following way:

- If \( T \) is a fundamental type, its associated set of namespaces is empty.
- If \( T \) is a class type, its associated namespaces are the namespaces in which the class and its direct and indirect base classes are defined.
- If \( T \) is a union or enumeration type, its associated namespace is the namespace in which it is defined.
- If \( T \) is a pointer to \( U \), a reference to \( U \), or an array of \( U \), its associated namespaces are the namespaces associated with \( U \).
- If \( T \) is a pointer to function type, its associated namespaces are the namespaces associated with the function parameter types and the namespaces associated with the return type. [recursive]

Koenig Lookup

Example

   namespace NS
   {
       class A {};
       void f( A ) {};
   }

   int main()
   {
       NS::A a;
       f( a );  //calls NS::f
   }
Example

```cpp
#include <iostream>

int main()
{
    // Where does operator<<() come from?
    std::cout << "Hello, World" << std::endl;
    return 0;
}
```

Include directives = multiple declarations

```cpp
#include "queue.h" // as if declaration were textually present
void f () { ... }

#include "queue.h" // second declaration in different client
void g () { ... }
```

- headers are safer than cut-and-paste, but not as good as a proper module system

Linking

- an external declaration for a variable indicates that the entity is defined elsewhere
  ```cpp
extern int x; // will be found later
  ```

- a function declaration indicates that the body is defined elsewhere

- multiple declarations may denote the same entity
  ```cpp
extern int x; // in some other file
  ```

- an entity can only be `defined` once

- missing/multiple definitions cannot be detected by the compiler: they result in link-time errors

Modules in JAVA

- package structure parallels file system

- a package is a directory

- a class is compiled into a separate object file

- each class declares the package in which it appears

  ```java
  package polynomials;
  class poly {
      ... // in file .../alg/polynomials/poly.java
  }
  ```

  ```java
  package polynomials;
  class iterator {
      ... // in file .../alg/polynomials/iterator.java
  }
  ```

Default: anonymous package in current directory.
Dependencies between classes

- dependencies indicated with `import` statements:
  
  ```java
  import java.awt.Rectangle; // declared in java.awt
  
  import java.awt.*; // import all classes // in package
  ```

- no syntactic sugar across packages: use expanded names or import

- none needed in same package: all classes in package are directly visible to each other

Modules in ML

There are three entities:

- **signature**: an interface
- **structure**: an implementation
- **functor**: a parameterized structure

A structure implements a signature if it defines everything mentioned in the signature (in the correct way).

---

ML signature

An ML *signature* specifies an interface for a module.

```ml
signature STACKS =
sig
  type stack
  exception Underflow
  val empty : stack
  val push : char * stack -> stack
  val pop : stack -> char * stack
  val isEmpty : stack -> bool
end
```

---

ML structure

A *structure* provides an implementation.

```ml
structure Stacks : STACKS =
struct
  type stack = char list
  exception Underflow
  val empty = [ ]
  val push = op:::
  fun pop (c::cs) = (c, cs)
  | pop [] = raise Underflow
  fun isEmpty [] = true
  | isEmpty _ = false
end
```
Comparisons

<table>
<thead>
<tr>
<th></th>
<th>ADA</th>
<th>C++</th>
<th>JAVA</th>
<th>ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>used to avoid name clashes</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>access control</td>
<td>✔</td>
<td>weak</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>is closed</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
</tr>
</tbody>
</table>

Relation between interface and implementation:

- **ADA**:
  - one package (interface) ⇔ one package body
- **ML**:
  - one signature *can be implemented by* many structures
  - one structure *can implement* many signatures

---

Programming Assignment 2

**Write a better SAT solver in ML**

- In assignment 1, you used exhaustive enumeration of assignments to test satisfiability
- In this assignment, you will implement one of two variations of the classical Davis-Putnam algorithm

Both of these algorithms are decision procedures for satisfiability of propositional formulas in CNF.

The first algorithm, Davis-Putnam (DP) was published in 1960, and is often confused with the later, more popular algorithm presented by Davis, Logemann, and Loveland in 1962, which we will refer to as Davis-Putnam-Logemann-Loveland (DPLL).

---

**Davis-Putnam Algorithm**

There are three satisfiability-preserving transformations in DP.

- The 1-literal rule
- The affirmative-negative rule
- The rule for eliminating atomic formulas

The first two steps reduce the total number of literals in the formula.

The last step reduces the number of variables in the formula.

By repeatedly applying these rules, eventually we obtain a formula containing an empty clause, indicating unsatisfiability, or a formula with no clauses, indicating satisfiability.

---

**Davis-Putnam Algorithm**

**The 1-literal rule**

Also called *unit propagation*.

Suppose \((p)\) is a unit clause (clause containing only one literal). Let \(\neg p\) denote the negation of \(p\) where double negation is collapsed (i.e. \(\neg \neg q \equiv q\)).

- Remove any instances of \(\neg p\) from the formula.
- Remove all clauses containing \(p\) (including the unit clause itself).
**Davis-Putnam Algorithm**

The affirmative-negative rule

Also called pure literal rule.

If a literal appears only positively or only negatively, delete all clauses containing that literal.

*Why does this preserve satisfiability?*

**DPLL Algorithm**

In the worst case, the resolution rule can cause a quadratic expansion every time it is applied.

For large formulas, this can quickly exhaust the available memory.

The DPLL algorithm replaces resolution with a splitting rule.

- Choose a propositional symbol $p$ occurring in the formula.
- Let $\Delta$ be the current set of clauses.
- Test the satisfiability of $\Delta \cup \{p\}$.
- If satisfiable, return true.
- Otherwise, return the result of testing $\Delta \cup \{\neg p\}$ for satisfiability.

**Example**

$$0 \parallel 1 \lor 2, \quad \overline{1} \lor \overline{2}, \quad 2 \lor 3, \quad \overline{3} \lor 2, \quad 1 \lor 4$$
Example

∅ | 1 ∨ 2, 1 ∨ 2, 2 ∨ 3, 3 ∨ 2, 1 ∨ 4 =⇒ (PureLiteral)
4 | 1 ∨ 2, 1 ∨ 2, 2 ∨ 3, 3 ∨ 2, 1 ∨ 4 =⇒ (Decide)
4 1^d | 1 ∨ 2, 1 ∨ 2, 2 ∨ 3, 3 ∨ 2, 1 ∨ 4 =⇒ (UnitProp)
4 1^d 2 | 1 ∨ 2, 1 ∨ 2, 2 ∨ 3, 3 ∨ 2, 1 ∨ 4

Example

∅ | 1 ∨ 2, 1 ∨ 2, 2 ∨ 3, 3 ∨ 2, 1 ∨ 4 =⇒ (PureLiteral)
4 | 1 ∨ 2, 1 ∨ 2, 2 ∨ 3, 3 ∨ 2, 1 ∨ 4 =⇒ (Decide)
4 1^d | 1 ∨ 2, 1 ∨ 2, 2 ∨ 3, 3 ∨ 2, 1 ∨ 4 =⇒ (UnitProp)
4 1^d 2 | 1 ∨ 2, 1 ∨ 2, 2 ∨ 3, 3 ∨ 2, 1 ∨ 4
Example

|   | \( \emptyset \) | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (PureLiteral) |
|---|-----------------|--------------------------------------------------------|
| 4 | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (Decide) |
| 4 1\textsuperscript{st} | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (UnitProp) |
| 4 1\textsuperscript{st} 2 | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (UnitProp) |
| 4 1\textsuperscript{st} 3 | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (Backtrack) |
| \( 4 \top \) | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (Backtrack) |
| 4 \( \top \) 2 | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (Fail) |

\( \text{fail} \)

\( \emptyset \) | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (PureLiteral) |
| 4 | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (Decide) |
| 4 1\textsuperscript{st} | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (UnitProp) |
| 4 1\textsuperscript{st} 2 | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (UnitProp) |
| 4 1\textsuperscript{st} 3 | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (Backtrack) |
| \( 4 \top \) | \( 1 \lor 2 \), \( \neg 1 \lor 2 \), \( 2 \lor 3 \), \( \neg 3 \lor 2 \), \( 1 \lor 4 \) | \( \Rightarrow \) (UnitProp) |

\( \text{fail} \)

Result: **Unsatisfiable**