# Homework 5

Due 10 November

Handout 6
CS242: Autumn 2004
3 November

## Reading

1. Read Chapter 9, Data Abstraction and Modularity, skipping section 9.2.5 on datatype induction and the material on ML modules and CLU abstraction mechanisms (unless you are interested).
2. Read Chapter 10, Concepts in Object-Oriented Languages.
3. Read Chapter 11 on Simula and Smalltalk.

## Problems

### 1. Efficiency vs. Modularity

The text describes the following form of heapsort:

```
function heapsort1(n:int, A:array[1..n])
begin
   s := empty;
   for i := 1 to n do       s := insert(A[i], s);
   for i := n downto 1 do   (A[i],s) := deletemax(s);
end
```

where heaps (also known as priority queues) have the following signature

```
empty : heap
insert : elt * heap -> heap
deletemax: heap -> elt * heap
```

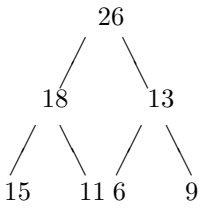
and specification:

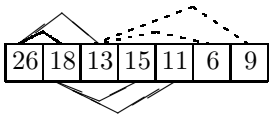
- Each heap has an associated multiset of elements. There is an ordering $<$ on the elements that may be placed in a heap.
- `empty` has no elements.
- `insert(elt,heap)` returns a heap whose elements are `elt` plus the elements of `heap`.
- `deletemax(heap)` returns an element of `heap` that is $\geq$ all other elements of `heap`, together with a heap obtained by removing this element. If `heap` is empty, `deletemax(heap)` raises an exception.

This problem asks you to reformulate the signature of heaps and the sorting algorithm so that the algorithm is more efficient, without destroying the separation between the sorting algorithm and the implementation of heaps. You may assume in this problem that we use pass-by-reference everywhere.

The standard non-modular heapsort algorithm uses a clever way of representing a binary tree by an array. Specifically, we can think of an array `A` of length `n` as a tree by regarding $\texttt{A}[1]$ as the root of the tree, and elements $\texttt{A}[2*\texttt{k}]$ and $\texttt{A}[2*\texttt{k}+1]$ as the left and right children of $\texttt{A}[\texttt{k}]$. One efficient aspect of this representation is that the the links between tree nodes do not need do be stored; we only need memory locations for the data stored at the tree nodes.

We say a binary tree is a *heap* if the value of the root of any subtree is the maximum of all the node values in that subtree. For example, the following tree is a heap, and can be represented by the array next to it.




The standard heapsort algorithm has the form

```
function heapsort2(n:int, A:array[1..n])
begin
   variable heap_size:int  :=  n;
   build_heap(n,A);
   for i := n downto 2 do
      swap(A[1], A[i]);
      heap_size := heap_size - 1;
      heapify(heap_size,A);
end
```

where the procedure `build_heap(n,A)` reorders the elements of array `A` so that they form a heap (binary search tree), and procedure `heapify(k,A)` restores array elements $A[1], \ldots, A[k]$ to heap form, assuming that only $A[1]$ was out of place.

(a) Assume that procedures `insert`, `deletemax`, and `heapify` all take time $O(\log n)$, i.e., time proportional to the logarithm of the number of elements in the heap, and `build_heap` takes time $O(n)$, i.e., time linear in the number of elements in the heap.* Compare the time and space requirements of the two algorithms, `heapsort1` and `heapsort2`, and explain what circumstances might make you want to choose one over the other.

(b) Suppose we change the signature of heaps to

```
make_heap : array * int -> heap
deletemax: heap -> elt
```

where `deletemax` may have a side-effect on the heap, and there is no longer `empty` or `insert`.

In addition to the specification of a return value, which is needed for a normal function, a function `f` with side effects should specify the before and after values of any variables that change, using the following format:

`f`: If $\texttt{x}_{before}$ is ... then $\texttt{x}_{after}$ is ...

The idea here is that the behavior of a function that may change the values of its arguments can be specified by describing the relationship between values before the call and values after the call. This "if ... then ..." form also let's you state any preconditions you might need, such as that the array `A` has at least `n` elements.

Write a specification for this modified version of heaps.

(c) Explain in words (or some kind of pseudo-code if you prefer) how you might implement the modified form of heap with imperative operations described in part 1b. You may assume that procedures such as `build_heap`, `heapify`, and `swap` are available. Assume that you will use your heaps for some form of heapsort. Try to make your operations efficient, at least for this application if not for all uses of heaps in general.

(*Hint:* you will need to specify a representation for heaps and an implementation of each function. Try representing a heap by a pair $\langle i, A \rangle$, where $A$ is an array and $0 < i \leq length(A)$.)

*Function `build_heap` works by a form of iterated insertion. This might require $O(n \log n)$ but analysis of the actual code for `heapify` allows us to show that it takes $O(n)$ time. If this interests or puzzles you, see *Introduction to Algorithms,* by Cormen, Leiserson, and Rivest, section 7.3.

(d) Write a heapsort algorithm (`heapsort3`) using the modified form of heap with imperative operations described in part 1b. How will the time and space compare with the other two algorithms?

(e) In all likelihood, your algorithm in part 1d (`heapsort3`) is not as time- and space-efficient as the standard heapsort algorithm (`heapsort2`). See if you can think of some way of modifying the definition or implementation of heaps that would let you preserve modularity and meet the optimum efficiency for this algorithm, or explain some of the obstacles for achieving this goal. (Alternatively, argue that your algorithm *is* as efficient as the standard heapsort.)

## 2. Function Templates and Overloading

As we discussed in connection with polymorphism, function templates are a way to add type variables to the C++ type system. In the example below, the type `T` allows the function `f` to take different types of arguments without changing the function body. The declarations on lines (a)–(c) are referred to as *function base templates*:

```
template<class T> void f( T );              // (a)
template<class T> void f( int, T, double ); // (b)
template<class T> void f( T* );             // (c)
void f( double );                           // (d)

int main() {
long l;
int i;
double d;

f( l );        // calls (a) with T = long
f( i, 42, d ); // calls (b) with T = int
f( &i );       // calls (c) with T = int
f( i );        // calls (a) with T = int
f( d );        // calls (d)
}
```

The complication with function templates and overloading arises when we have multiple versions of a function that can match a single call. In the absence of templates, the C++ compiler prevents this sort of clash. With templates, though, we can see that the declarations (a) and (d) clash, but the C++ compiler does not complain about this code. For example, the call `f( d )` on the last line of `main` could resolve to either declaration (a) or declaration (d).

Here are some rules for resolving overloaded function templates:

(a) Nontemplate functions are preferred. If a nontemplate function matches the parameter types as well as any function template, the nontemplate function is used.

(b) If there are no nontemplate functions that are at least as good, then a function base template will be used. Which function base template gets selected depends on which matches best and is the "most specialized," according to a set of fairly arcane rules:

   i. If there is one "most specialized" function base template, that one gets used.

   ii. If there is a tie for the "most specialized" function base template, the call is ambiguous because the compiler cannot decide which is a better match. The programmer will have to do something to qualify the call and say which one is wanted.

   iii. If there is no function base template that can be made to match, the call is bad and the programmer will have to fix the code.

We can provide an intuitive definition of "specialization" by saying that one template is more specialized than another if it can match fewer types. For example, the function base template on line (c) is more specialized than the function base template on line (a).

(a) (5 points) For each of the calls in the example above, write down which of the rules above is used to resolve clashes in overload resolution. If there is no clash (i.e. only one choice among (a)-(d) is possible), write "no clash." Otherwise write the specific rule that is used (e.g., (b)-iii, would refer to the very last rule).

(b) Given that the call to `f(d)` could be handled by the declaration on line (a), why might a programmer choose to overload `f` with the declaration on line (d)? More specifically, write a body for the declaration on line (a) that contains no more than one line and will cause the C++ type checker to report an error if you eliminated the declaration on line (d). In writing your definition, remember that C and C++ will often implicitly cast the parameters to an operator.

```
template<class T> void f(T x) { _____________________________; }
```

## 3. Functions expressions using overloading

The Boost Lambda Library (BLL) is a C++ template library that implements a form of lambda expression in C++, allowing a programmer to define unnamed function within C++ statements or function calls. Here's a short example:

```
for_each(a.begin(), a.end(), std::cout << _1 << ' ');
```

The expression `std::cout << _1 << ' '` defines a unary (one-argument) function object, with `_1` indicating the parameter of this function. Within each iteration of the loop inside the function `for_each`, the unnamed function is called with an element of the collection `a` as the actual argument, and the function body is evaluated with the actual argument substituted for the parameter `_1`. A three-argument function is defined using `_1`, `_2`, and `_3`, for example. The maximum number of arguments to any BLL lambda expression is 9.

A BLL function may be called using the usual C syntax. For example, `std::cout << _1 << ' '` can be applied to the argument 5 by writing `(std::cout << _1 << ' ')(5)`. A function call with too many arguments is a compile-time error.

The implementation of BLL uses overloading. For example, `_1` by itself is a lambda expression, and `cout << _1` is a lambda expression because << is overloaded so that if one of its operands is a lambda expression, the result is a lambda expression. The lambda expression `_1` defines the function with `(_1)(x) = x` for any argument value `x`.

(a) Write an expression in BLL syntax for the function $f(x, y) = x + y$, which would be written `(lambda (x y) (+ x y))` in Lisp.

(b) Similarly, write a BLL version of the Lisp expression:
`((lambda (f x) (f (+ x 1))) (lambda (y) (+ y 2)) 3)`

(c) Consider the operator +. Describe in a few sentences how you think BLL changes or adds to the definition of + so that

i. `_1 + 2` and `2 + _1` are lambda expressions,

ii. `(_2 + 2)(x,y)` correctly compiles as a function definition and call, and

iii. `(_3 + 2)(x,y)` results in a compile-time error.

## 4. Expression Objects

We can represent expressions given by the grammar

$$e \quad ::= \quad num \mid e + e$$

using objects from a class called `expression`. We begin with an "abstract class" called expression. While this class has no instances, it lists the operations commons to all kinds of expressions. These are a predicate telling whether there are subexpressions, the left and right subexpressions (if the expression is not atomic), and a method computing the value of the expression.

```
class expression() =
    private fields:
        (* none appear in the _interface_ *)
    public methods:
        atomic?()  (* returns true if no subexpressions *)
        lsub()     (* returns ''left'' subexpression if not atomic *)
        rsub()     (* returns ''right'' subexpression if not atomic *)
        value()    (* compute value of expression *)
end
```

Since the grammar gives two cases, we have two subclasses of `expression`, one for numbers and one for sums.

```
class number(n) = extend expression() with
    private fields:
        val num = n
    public methods:
        atomic?() = true
        lsub   () = none  (* not allowed to call this, *)
        rsub   () = none  (* because atomic?() returns true *)
        value  () = num
end

class sum(e1,e2) = extend expression() with
    private fields:
        val left = e1
        val right = e2
    public methods:
        atomic?() = false
        lsub   () = left
        rsub   () = right
        value  () = ( left.value() ) + ( right.value() )
end
```

(a) *Product Class*

Extend this class hierarchy by writing a `prod` class to represent product expressions of the form

$$e \quad ::= \quad \ldots \mid e * e$$

(b) *Method Calls*

Suppose we construct a compound expression by

```
val a = number(3);
val b = number(5);
val c = number(7);
val d = sum(a,b);
val e = prod(d,c);
```

and send the message `value` to `e`. Explain the sequence of calls that are used to compute the value of this expression: `e.value()`. What value is returned?

(c) *Unary Expressions*

Extend this class hierarchy by writing a `square` class to represent squaring expressions of the form

$$e \quad ::= \quad \ldots \mid e^2$$

What changes will be required in the `expression` interface? What changes will be required in subclasses of `expression`? What changes will be required in functions that use `expressions`?[†] What changes will be required in functions that do not use `expressions`? (Try to make as few changes as possible to the program.)

```
Hint :

consider an existing user function that uses expression class: this
function will return the number of leaf.

fun count (e)
   var i:=0
   if e.atomic?() then i:= 1
   if not e.atomic?()then i:= count(e.lsub()) + count (e.rsub())
   return i
end


consider also the following square class:

class square (e1) = extend expression() with
   private fields:
      val left = e1
   public methods:
      atomic?() = true
      lsub() = left
      rsub() = left
      value() = left.value() * left.value()


this square class will lead to error, for instance, let e =
1+2+sq(3) if we pass this to the count function, it will return 4 as
oppose to 3. so in this case the user function breaks. if we modify
the rsub() = none in the square class, the user function will also
breaks.

to fix the problem, we need to add another method in expression
class to distinguish between atomic, unary and binary. also the user
function that uses expression must be changed so that it aware of
unary expression.
```

(d) *Ternary Expressions*

Extend this class hierarchy by writing a `cond` class to represent conditionals[‡] of the form

$$e \quad ::= \quad \ldots \mid e?e : e$$

What changes will be required if we wish to add this ternary operator? (As in part (c), try to make as few changes as possible to the program.)

(e) *N-Ary Expressions*

Explain what kind of interface to expressions we would need if we would like to support atomic, unary, binary, ternary and $n-$ary operators without making further changes to the

[†]Keep in mind that not all functions simply want to evaluate entire expressions. They may call the other methods as well.

[‡]In C, conditional expressions `a?b:c` evaluate `a`, and then return the value of `b` if `a` is non zero, or return the value of `c` if `a` is zero.

interface. In this part of the problem, we are not concerned with minimizing the changes to the program; instead, we are interested in minimizing the changes that may be needed in the future.

## 5. ..... Simula Inheritance and Access Links

In Simula, a class is a procedure that returns a pointer to its activation record. Simula prefixed classes are a precursor to C++ derived classes, providing a form of inheritance. This question asks about how inheritance might work in an early version Simula, assuming that the standard static scoping mechanism associated with activation records is used to link the derived class part of an object with the base class part of the object.

Sample `Point` and `ColorPt` classes are given in the text. For the purpose of this problem, assume that if `cp` is a `ColorPt` object, consisting of a `Point` activation record followed by a `ColorPt` activation record, the access link of the parent class (`Point`) activation record points to the activation record of the scope in which the class declaration occurs, and the access link of the child class (`ColorPt`) activation record points to activation record of the parent class.

(a) Fill in the missing information in the following activation records, created by executing the following code:

```
ref(Point) r;
ref(ColorPt) cp;
r :- new Point(2.7, 4.2);
cp :- new ColorPt(3.6, 4.9, red);
cp.distance(r);
```

Remember that function values are represented by closures, and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code. Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link." The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.

*Activation Records* | *Closures* | *Compiled Code*

| | | |
|---|---|---|
| (0) | r | • |
| | cp | • |
| (1) Point(. . . ) | access link | ( 0 ) |
| | x | |
| | y | |
| | equals | • |
| | distance | • |
| (2) Point part of cp | access link | ( 0 ) |
| | x | |
| | y | |
| | equals | • |
| | distance | • |
| (3) ColorPt(. . . ) | access link | ( ) |
| | c | |
| | equals | • |
| (4) cp.distance(r) | access link | ( ) |
| | q | ( *r* ) |

⟨ ( ), • ⟩ — code for equals

⟨ ( ), • ⟩

⟨ ( ), • ⟩ — code for distance

⟨ ( ), • ⟩

⟨ ( ), • ⟩ — code for cpt equals

(b) The body of `distance` contains the expression

$$\texttt{sqrt}((\texttt{x} - \texttt{q.x}) * *2 + (\texttt{y} - \texttt{q.y}) * *2)$$

which compares the coordinates of the point containing this `distance` procedure to the coordinate of the point `q` passed as an argument. Explain how the value of `x` is found when `cp.distance(r)` is executed. Mention specific pointers in your diagram. What value of `x` is used?

(c) This illustration shows that a reference `cp` to a colored point object points to the `ColorPt` part of the object. Assuming this implementation, explain how the expression `cp.x` can be evaluated. Explain the steps used to find the right `x` value on the stack, starting by following the pointer `cp` to activation record (3).

(d) Explain why the call `cp.distance(r)` only needs access to the `Point` part of `cp` and not the `ColorPt` part of `cp`.

(e) If you were implementing Simula, would you place the activation records representing objects `r` and `cp` on the stack, as shown here? Explain briefly why you might consider allocating memory for them elsewhere.

## 6. Smalltalk Run-time Structures

Here is a Smalltalk `Point` class whose instances represents points in the two-dimensional Cartesian plane. In addition to accessing instance variables, an instance method allows point objects to be added together.

| class name | Point |
|---|---|
| superclass | Object |
| class variables | *comment: none* |
| instance variables | x y |
| class messages and methods | *comment: instance creation*<br>**newX: xValue Y: yValue** \| \|<br>↑ self new x: xValue y: yValue |
| instance messages and methods | *comment: accessing instance vars*<br>**x: xCoordinate y: yCoordinate** \| \|<br>x ← xCoordinate<br>y ← yCoordinate<br>**x** \| \| ↑ x<br>**y** \| \| ↑ y<br>*comment: arithmetic*<br>**+ aPoint** \| \|<br>↑ Point newX: (x + aPoint x) Y: (y + aPoint y) |

(a) Complete the top half of the drawing of the Smalltalk run-time structure shown in Figure 1 for a point object with coordinates $(3, 4)$ and its class. Label each of the parts of the top half of the figure, adding to the drawing as needed.

(b) A Smalltalk programmer has access to a library containing the `Point` class, but she cannot modify the `Point` class code. In her program, she wants to be able to create points using either cartesian or polar coordinates, and she wants to calculate both the polar coordinates (radius and angle) and the Cartesian coordinates of points. Given a point $(x, y)$ in cartesian coordinates, the radius is `((x * x) + (y * y)) squareRoot`, and the angle is `(x/y) arctan`. Given a point $(r, \theta)$ in polar coordinates, the $x$ coordinate is `r * (`$\theta$ `cos)` and the $y$ coordinate is `r * (`$\theta$ `sin)`

i. Write out a subclass, PolarPoint, of Point and explain how this solves the programming problem. 4in

ii. Which parts of Point could you reuse and which would you have to define differently for PolarPoint? 3in

(c) Complete the drawing of the Smalltalk run-time structure by adding a `PolarPoint` object and its class to the bottom half of the figure you already filled in with `Point` structures. Label each of the parts and add to the drawing as needed.

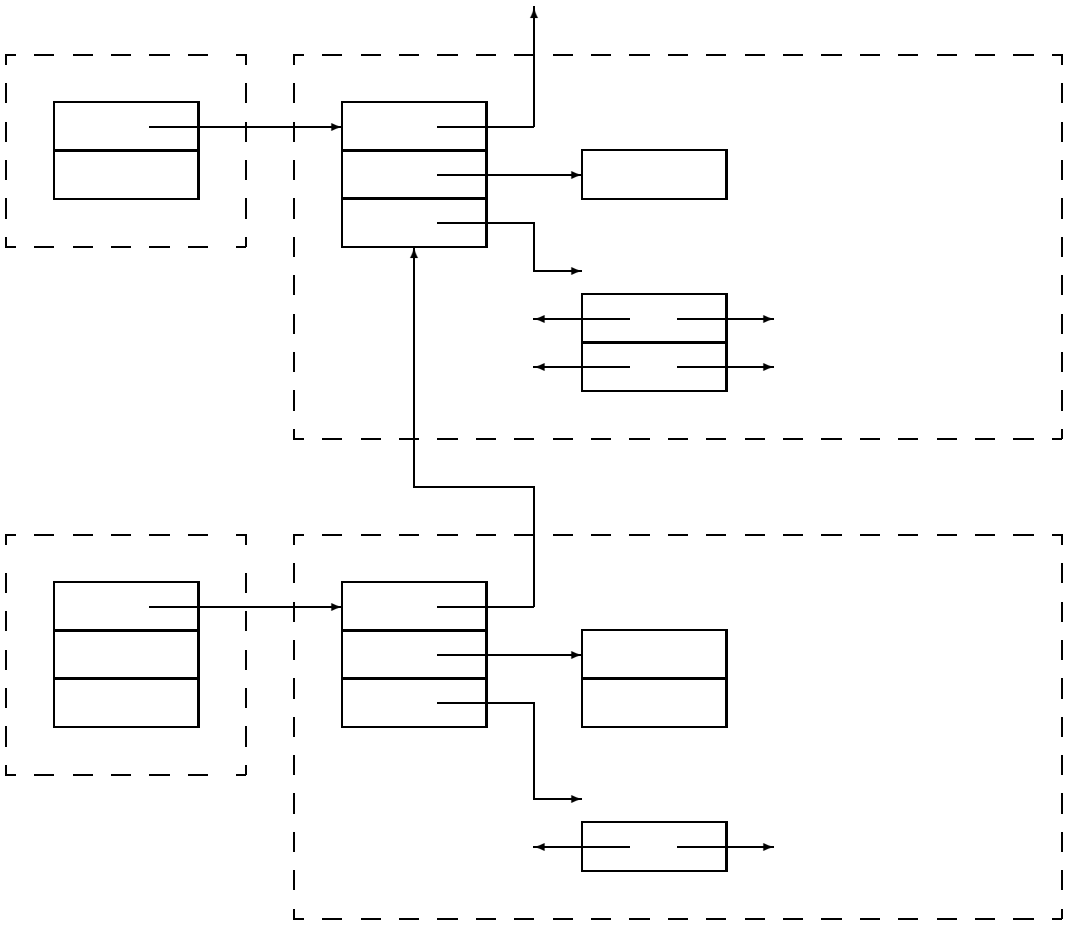


Figure 1: Smalltalk Run-Time Structures for Point and PolarPoint