Exercises 11 (Procedural abstraction) – Solutions

11A. *(C function declarations)*

Possible modified syntax:

decl = …
| *type id* '`(`' *formals*? '`)`' '`=`' *expr*

This change, in isolation, would reduce the expressive power of functions because they could not declare or use local variables, and they could not employ iteration.

To compensate, the language's repertoire of expressions should be expanded to include block-expressions and iterative expressions (e.g., array comprehensions).

11B. *(C functions and proper procedures)*

Possible modified syntax:

decl = …
| '`func`' *type id* '`(`' *formals*? '`)`' '`=`' *expr*
| '`proc`' *id* '`(`' *formals*? '`)`' *block-com*

expr = …
| *id* '`(`' *actuals*? '`)`'

com = …
| *id* '`(`' *actuals*? '`)`' '`;`'

11C. *(Copy vs reference parameters)*

(a) If the procedure call "`add(a, b, c);`" uses *copy* mechanisms:

On entry to the procedure, local variable `v1` is created and initialized with the value of `a`, local variable `v2` is created and initialized with the value of `b`, and local variable `s` is created but not initialized.

On exit from the procedure, the final value of `s` is assigned to the variable `c`; and all three local variables are destroyed.

(b) If the procedure call "`add(a, b, c);`" uses *reference* mechanisms:

On entry to the procedure, `v1` is bound to the value of `a`, `v2` is bound to the value of `b`, and `s` is bound to the variable `c`.

On exit from the procedure, all three bindings are discarded.

(c) In (a), the procedure computes the vector sum of `a` and `b` in the local variable `s`, then assigns it to `c`. In (b), the procedure computes the vector sum of `a` and `b` in `c` (since `s` refers to `c`). The net effect is the same, except that (a) entails more copying.

Exercises 12 (Data abstraction) – Solutions

12A. *(Programming without data abstraction)*

Programming a dictionary "package" in C:

(a) Write a header file named (say) `dict.h`:

```
int contains (char * word);
void remember (char * word);
```

Also write a compilation unit named `dict.c`:

```
#include "dict.h"
char ** words = …;  // some suitable data structure
int contains (char * word) ) {
  …  // search for word in words, returning 0 if not found
}
void add (char * word) ) {
  …  // add word to words
}
```

Each compilation unit containing application code must also include `trig.h`, as follows:

```
#include "dict.h"
void main (int argcount; char ** args) {
  …
  add("aardvark");
  …
  if (contains(args[i])) …
}
```

(b) This style of programming relies heavily on consistent inclusion of header files. In (a), if the application code did *not* include `dict.h`, any type errors in calls to `remember` and `contains` would be missed by the compiler.

12B. *(Abstract data type for complex numbers)*

```
class Complex {
  private float x, y;
  // … represents a complex number whose real part is x and
  // whose imaginary part is y.
  public Complex (float real, float imag) {
    x = real; y = imag;
  }
  public float mag () {
    return Math.sqrt(x*x + y*y);
  }
  public Complex plus (Complex that) {
    return new Complex(this.x+that.x, this.y+that.y);
  }
  public Complex minus (Complex that) {
    return new Complex(this.x-that.x, this.y-that.y);;
  }
  public Complex times (Complex that) {
    return new Complex(this.x*that.x - this.y*that.y,
          this.x*that.y + this.y*that.x);
  }
}
```

12C. *(Classes for geometric shapes)*

(a) Classes for straight lines and text boxes:

```
class Line extends Shape {
  private float l, o;  // length, orientation
  public Line (float l, float a)
  { … }
  public float length ()
  { return l; }
  public float orientation ()
  { return o; }
  public void draw ()
  { … }
}
class TextBox extends Box {
  private String t;  // text content
  public TextBox (int w, int h, String t)
  { … }
  public String content ()
  { return t; }
  public void draw ()
  { … }
}
```

(b) Declaration of a `Picture` class:

```
class Picture {
  private List<Shape> shapes;
  public Picture ()
  { shapes = new List<Shape>(); }
  public void add (Shape s)
  { shapes.add(s); }
  public void draw ()
  { for (Shape s : shapes) s.draw(); }
}
```

Exercises 13 (Generic abstraction) – Solutions

13A. *(Generic class for lists)*

Instantiating `List<T>` to declare a variable `itinerary` that will contain a list of airports:

```
List<Airport> itinerary;
```

13B. *(Generic class for priority queues)*

(a) Modified declaration of class `Fault`:

```
class Fault implements Comparable<Fault> {
// A Fault object is a date-stamped fault report.
  public Date date;
  public String description;
  …
  public int compareTo (Fault that)
  { return this.date.compareTo(that.date); }
}
```

(b) Instantiating the generic class to declare a variable `log`, which will contain a priority queue of fault reports ordered by date:

```
PQueue<Fault> log;
```

13C. *(Generic class for binary relations)*

(a) Declaration of generic class `Relation<X,Y>`:

```
class Relation<X,Y> {
// A Relation<X,Y> object is a set of pairs (x, y) such that
// the x values are of type X and the y values are of type Y.
  private …;
  public Relation ()
  { … }
  public void add (X x, Y y)
  { … }
  public boolean contains (X x, Y y)
  { … }
  public Set<Y> retrieve (X x)
  { … }
}
```

In order to implement the `contains` and `retrieve` methods, we must assume that the types `X` and `Y` are equipped with `equals` methods. This is not a problem, because all objects are equipped with such methods.

(b) Modified declaration of generic class `Relation<X,Y>`:

```
class Relation<X implements Comparable<X>, Y> {
  …  // as above
  public void display ()
  { … }
}
```