Programming Language Concepts:
Lecture Notes

Madhavan Mukund
Chennai Mathematical Institute
92 G N Chetty Road
Chennai 600 017
http://www.cmi.ac.in/~madhavan

Contents

Part I

Object-oriented programming

Chapter 1

Data encapsulation with classes

The starting point of object-oriented programming is to provide a more faithful implementation of the notion of "type" in programming. The programming languages we are familiar with come with standard built-in types that we can assign to variables and values:

For example, in C we can use `int`, `float`, `double`, `char`, ... In Haskell, similarly, we can use `Int`, `Float`, `Char`, `Bool`, ...

In addition, programming languages provide a way to assign a single name to a collection of values of the same type. The nature of this collection is determined by the underlying architecture assumed by the programming language. Since C reflects the standard von Neumann stored program architecture, we have arrays that provide "random" (i.e. equal time) access to each element of the collection, while Haskell, whose semantics is defined with respect to an idealized functional programming architecture, provides us with lists, which have "sequential access" (getting to the ith item takes time proportional to i) but which come with a number of useful decomposition functions that make inductive definitions easy to implement.

However, no programming language can expect to predefine all the useful "types" that we might need in a particular application—we might want to work with stacks or binary search trees or directed acyclic graphs or ... Typically, we define these in terms of the existing primitives available. For instance, in Haskell we might represent a stack as a list or, for efficiency, as a pair of lists, while in C we might represent a stack as an array or, if we want to let the stack grow arbitrarily large, as a linked list.

Regardless of the implementation, we expect a stack to be used in a particular way. Additions to the stack are done using a function "push", deletions by a function "pop" and the only enquiry that we can make about the state of the stack is the question "is-the-stack-empty". Suppose the stack is defined as

```
  int s[100], top_of_stack = 0;
```

in C and the stack grows as `s[0]`, `s[1]`, ... When the stack has 10 elements (i.e., `top_of_stack == 10`) we do not want a function to access, say, `s[7]`.

In a language like C, this “integrity” of the datatype stack is difficult to enforce. It can, in general, be achieved only through disciplined programming—there is no way to *guarantee* that a stack is never misused.

The preceding discussion on stacks applies equally to other kinds of complex datatypes like binary search trees, directed acyclic graphs etc. In the jargon of programming language theory, these are all examples of *Abstract Data Types*—data structures that have certain fixed functions to manipulate them and these functions are expected to be rigidly adhered to as the only means of accessing the underlying data, regardless of the implementation.

1.1 Classes

To facilitate the definition of abstract data types, the programming language Simula (1967) introduced a concept called a *class*. (The word class is not important in itself, except that it has stuck and so is now a standard term in the field.)

A class is a “type definition” consisting of two parts:

1. How the data is stored in this type.

2. What functions are available to manipulate this data.

For instance, we might represent a stack of integers by the following class definition:

```
class stack {
  int values[100];       /* values are stored in an array */
  int tos = 0;           /* top of stack, initialize to 0 */

  push (int i, ...){     /* push i onto stack */
    values[tos] = i;
    tos = tos+1;         /* Should check that tos < 100!! */
  }

  int pop (...){         /* pop top of stack and return it */
    tos = tos - 1;       /* Should check that tos > 0!! */
    return values[tos];
  }

  bool is_empty (...){  /* is the stack empty? */
    return (tos == 0);  /* yes iff tos is 0    */
  }
}
```

At some places in this hypothetical definition, the function parameters have been left unspecified and denoted “. . . ”. We will fill in these blanks shortly.

We can use the class name as a new type in the language and define variables of this type elsewhere in the program, as follows:

```
stack s,t;
```

These definitions only provide the capability for `s` and `t` to denote stacks. No storage is allocated at this point. They correspond to a pointer declaration of the type `int *p` in C, which gives `p` the ability to point to a memory location holding an integer but does not, in itself, allocate a valid memory location for `p` to point to.

In order to get a "real" stack, we have to ask for it. For instance, we might say:

```
s = new stack;
t = new stack;
```

This generates two independent stacks, one named `s` and the other `t`. If, instead, we had written

```
s = new stack;
t = s;
```

we get a single stack with two names, `s` and `t`, either of which can be used to access the stack.

The class definition only provides a *template* for a datatype. The operation `new` generates an *instance* of the template. The word *object* in object-oriented programming is synonymous with the phrase *instance of a class*.

How do we manipulate the objects `s` and `t`? Suppose we want to push the value 7 onto the stack `s`. We do this by writing:

```
s.push(7);
```

Note the difference in style with a conventional language. In C or Haskell, `push` would take two arguments, the stack itself and the value to be pushed. In the object-oriented framework, each instance of the class has its "own" copy of these functions and is thus an implicit argument to the function.

Similarly, to retrieve the value at the top of `s` into a local variable and to query whether `t` is empty, we would write the following, respectively:

```
i = s.top();
if (t.is_empty()) {...}
```

We can now say what should appear in place of the mysterious ... in the function definitions in our class stack—nothing! In other words, the functions `pop` and `is_empty` take no arguments at all—the only argument they need is the name of the stack on which to operate, which is implicitly given since these functions will be called with respect to a fixed stack. Similarly, `push` needs only one argument: the value to be pushed.

Is there any significant difference between a C style function `push(s,i)` which takes two arguments, one of which is the stack, and the object-oriented version `s.push(i)`, where the stack is implicitly passed? Arguably this is only a matter of style and the two approaches have the same "effect".

But, philosophically, the syntax reflects the object-oriented approach: when we create an object (an instance of a class) we are supposed to visualize the object as a black box with a display which can show us some information about the box and some slots where we can insert data. The box is equipped with "buttons", one for each function. Pushing a button invokes the corresponding function, with the input slots used to slip in parameters to the function, and the value returned shown on the display. Our access to the internal state of the object, whether to observe it or to modify it, is expected to be *only* via pushing buttons. (Note: More jargon—the functions defined in classes are often called *methods* and the data items are often called *instance variables* or *instance fields.*)

1.2 Data encapsulation (public and private)

What is the status of the instance variables defined within a class? It seems desirable that these variables should not be accessible externally. For instance, there should be no way to use the variables `s` and `t` of type `stack` to directly manipulate the contents of `tos` and the array `value` within `s` and `t`. For, if these internal variables are accessible externally, we once again have to deal with the problem of maintaining the integrity of our abstract data type!

One way to achieve total data integrity is to make all internal data variable inaccessible from outside. Of course, there is almost always a requirement to be able to access the values of some or all of the variables and also modify or update them—for example, consider the following definition:

```
class date {
   int day, month, year;
}
```

where one might reasonably expect to be able to find out what date is currently stored, as well as, perhaps, to reset the date to a fresh value.

To meet this requirement while keeping internal variables inaccessible, we could write functions like `get_date()` and `set_date(...)` that provide limited, controlled access to the data stored in the class. (Note: In object-oriented jargon, a method that reads the values of internal variables, like `get_date()`, is called an *accessor method* and a method that updates the values of internal variables, like `set_date(...)` is called a *mutator method.*) Using appropriate accessor and mutator functions, we can control very precisely the degree to which the data in the stack can be manipulated externally.

However, probably to accommodate lazy programmers, most object-oriented languages do permit direct external access to variables. The syntax is similar to that for accessing members of structures in C (or fields of records in Pascal). For example, the field `tos` in the stack `s` is designated `s.tos`, so we can write code like:

```
if (s.tos == 0){ ... }
```

Once we permit this facility, we have to introduce an additional mechanism to prevent misuse—that is, to retain data integrity. This is typically done by adding the qualifiers `public` and `private` to each variable defined in a class. As should be clear, a variable that is declared to be `public` is accessible from outside while one that is declared `private` is not. One would write

```
class stack{
  private int values[100];
  private int tos = 0;
  ...
}
```

to ensure that the only way to access the stack is via `push`, `pop` and `is_empty`. Of course, `private` only refers to the external world: the functions within the class like `push`, `pop` and `is_empty` have full access to all the data variables defined in the class.

What about methods—should all methods be accessible by default? Suppose our language has a mechanism to request storage at run-time (like the `malloc` function in C). We might then incorporate a method `extend_stack` into our class and use it when the stack becomes full:

```
class stack {
  ...
  push (int i){    /* push i onto stack */
    if (stack_full){
      extend_stack();
    }
    ...            /* Code to add i to stack * /
  }

  extend_stack(){
    ...  /* Code to get additional space for stack data */
  }
  ...
}
```

Clearly `extend_stack` is an implementation detail that is not intended to be revealed outside the class. It would make sense to extend the use of the attributes `public` and `private` to methods as well, making, in this case, only the methods `push`, `pop` and `is_empty` public and all other methods, including `extend_stack`, private.

1.3 Static components

A C program is a collection of functions, all at the same level and hence all on an equal footing. To unambiguously designate where the computation starts, we insist that there is always a function called `main` where execution begins.

In a purely object-oriented language, a program is a collection of classes. Every function and variable that is defined in the program must lie within some class. However, as we have seen, classes are just templates and their definitions come to life only after they are instantiated using `new`. Initially all classes are inactive and there is no way for the program to get started—effectively, the program is deadlocked.

So, there is a need to be able to define functions whose existence does not depend on a class being instantiated.

There is another reason for having such functions. Consider, for example, routine functions like `read` and `write` (or `scanf` and `printf`, in C terminology), or, for example, mathematical functions like `sin`, `sqrt`, ... Clearly, it makes no sense to have to artificially instantiate a class in order to use such functions.

One way to get around this problem without introducing free floating functions that live outside classes (as happens, for instance, in C++) is to add the qualifier `static`. A function that is marked static inside a class is available without having to instantiate the class. We could have a definition of the form

```
class IO {
  public static ...  read(...) { ... }
  public static ...  write(...) { ... }
  ...
}
```

and then invoke these functions from outside as `IO.read(...)` and `IO.write(...)`.

Static functions also solve the problem of where to start—like C, we can insist that the collection of classes includes one static function with a fixed name (say, `main`, like C) and begin execution by invoking this method.

In addition to static funtions, it also makes sense to have static data variables. This can be used to define constants, such as:

```
class Math {
  public static double PI = 3.1415927;
  public static double E  = 2.7182818;
  public static double sin(double x) { ... }
  ...
}
```

Notice that the designation `static` is orthogonal to `public`/`private`. Does it make sense to have something that is `private static`?

Though it appears useless at first sight, we could consider a class all of whose instances needs some "magic numbers" to do their job, but where these "magic numbers" need not be visible externally. Suppose we write a class that computes the annual interest yield on a fixed deposit. Typically, the rate varies depending on the length of the deposit, but it might be that all the variable rates are computed from a single base rate. We might then write

```
class interest-rate {
  private static double base_rate = 7.32;

  private double deposit-amount;

  public double threemonth-yield(){ ... }  /* uses base-rate
                                              and deposit-amount */
  public double sixmonth-yield(){ ... }    /* uses base-rate
                                              and deposit-amount */
  public double oneyear-yield(){ ... }     /* uses base-rate */
                                              and deposit-amount */
  ...
}
```

The idea is that *all* instances of interest-rate share a single copy of the static variable `base_rate`, so that there is no unnecessary duplication of data and also no inconsistency.

Static variables are shared across all instances of a class. Thus, we can also use static variables to keep track of global information such as how many times a given function is called across all instances of the class. For example, we could write

```
class stack {
  private int values[100];      /* values are stored in an array */
  private int tos = 0;          /* top of stack, initialize to 0 */

  private static int num_push = 0;  /* number of pushes across all
                                       stacks */

  push (int i, ...){    /* push i onto stack */
    values[tos] = i;
    tos = tos+1;        /* Should check that tos < 100!! */
    num_push++;         /* update static variable */
  }

  ...
}
```

Here again, it makes sense to have `num_push` restricted to be private because we want it to be incremented only by the method `push` within the class `stack`.

We have to be careful when we mix static and non-static entities in the same class. A non-static function is tied to a specific instance (object) and so can implicitly refer to the current state of the object as well as to static data (such as the yield functions above).

However, since a static function is shared amongst all instances of a class (and there need not even be a single instance in existence for the static function to be used), such a function should not make any reference to any non-static components of the class. We will address this point in more detail later.

1.4 Constants (the attribute final)

Consider our earlier definition

```
class Math {
  public static double PI = 3.1415927;
  ...
}
```

The intention was to make the value Math.PI available everywhere to use in expressions: e.g.,

```
area = Math.PI * radius * radius;
```

However, there is nothing to prevent the following improper update of the public static field `PI`:

```
Math.PI = 3.25;
```

To forbid this, we attach another attribute, indicating that a value but may not be altered. Following Java terminology, we use the notation `final`, so we would modify the definition above to say:

```
class Math {
  public static final double PI = 3.1415927;
  ...
}
```

and achieve what we set out to have—a publicly available constant that exists without instantiating any objects.

Actually, the modifier `final` has other ramifications. We will see these later, when we talk about inheritance.

1.5 Constructors

When we create an object using `new`, we get an uninitialized object. We then have to set the values of the instance variables to sensible values (almost always this has to be done via appropriate methods, because these variables will be private). It is natural to ask for the analogue of a declaration of the form:

```
int i = 10;
```

which "creates" a variable `i` of type `int` and also, simultaneously, sets its initial state.

In the object-oriented paradigm, this initialization is performed by special methods, called *constructors*. A constructor is a method that is called implicitly when the object is created. It *cannot* be called after the object has been created.

In Java, a constructor is written as a public method with no return value which takes the same name as the class. For example, we can define a class `Date` as follows:

```
class Date{

  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }

}
```

Now, if we write

```
Date d = new Date(27,1,2003);
```

`d` points to a new `Date` object whose values are initialized with `day = 27`, `month = 1` and `year = 2003`.

We may want to have more than one way to construct an object. If no year is supplied, we might set the field year to the current year, by default.

We can add a second constructor as follows:

```
public Date(int d, int m)
  day = d;
  month = m;
  year = 2003;
}
```

Now, if we write

```
Date d1 = new Date(27,1,2002);
Date d2 = new Date(27,1);
```

d1 calls the first constructor, as before, while d2 calls the second constructor (where year is set to 2003 by default).

This ability to have multiple constructors with different "signatures" extends to methods as well. In Java, the signature of a method is its name plus the types of its arguments. One can overload method names just like we have overloaded the constructors above. In fact, in the Java built-in class Arrays, there is a static method sort that sorts arbitrary arrays of scalars. In other words, we can write:

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);  // sorts contents of darr
Arrays.sort(iarr);  // sorts contents of iarr
```

This is done by defining, within the class Arrays, multiple methods named sort with different signatures, as follows:

```
class Arrays{
  ...
  public static void sort(double[] a){  // sorts arrays of double[]
    ...
  }
  public static void sort(int[] a){  // sorts arrays of int[]
    ...
  }
  ...
}
```

When we invoke Arrays.sort with an argument a, the type of a automatically determines which version of the overloaded method sort is used.

Coming back to constructors, what happens if we have no constructors defined? In this case, Java provides a "default" constructor that takes no arguments and sets all instance variables to some sensible defaults (e.g., an int is set to 0). Suppose we have a class as follows:

```
class no_constructor{
  private int i;

  // some methods below
  ...
}
```

We would then write something like:

```
no_constructor n = new no_constructor(); // Note the () after
                                         // the class name
```

However, if there is at least one constructor defined in the class, the default constructor is withdrawn. So, if we have the class `Date` as given above, it is an error to write:

```
Date d = new Date();
```

If we want this to work, we must explicitly add a new constructor that has no arguments.

Remember that it is not possible to invoke a constructor later. Though `Date(int,int,int)` is a public "method" in the class, it has a different interpretation. We cannot say;

```
Date d = new Date(27,1,2003);
...
d.Date(27,1,2003);
```

One constructor can call another, using the word `this`. We can rewrite the two constructors in the class `Date` as follows:

```
class Date{

  private int day, month, year;

  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }

  public Date(int d, int m){
    this(d,m,2003);
  }

}
```

The second constructor invokes the first one by supplying a fixed third argument. In Java, such an invocation using `this` *must* be the first statement in the constructor. We can reverse the constructors as follows:

```
class Date{

  private int day, month, year;
  public Date(int d, int m){
    day = d;
    month = m;
    year = 2003;
  }

  public Date(int d, int m, int y){
    this(d,m);   // this sets year to 2003
    year = y;    // reset year to the value supplied
  }

}
```

Chapter 2

Java Interlude 1 Quick introduction to Java

2.1 Structure of a Java program

A Java program is a collection of classes. Each class is normally written in a separate file and the name of the file is the name of the class contained in the file, with the extension `.java`. Thus, the class stack defined earlier would be stored in a file called `stack.java`.

As we noted in Chapter 1, we have to fix where the program starts to execute. Java requires that at least one of the classes has a public static method called main. More precisely we need a method with the following *signature*:

```
public static void main(String[] args)
```

We have already seen what `public` and `static` mean. The word `void` has the same connotation as in C — this function does not return a value. The argument to `main` is an array of strings (we'll look at strings and arrays in Java in more detail later)—this corresponds to the `argv` argument to the main function in a C program. We don't need `argc` because in Java, unlike in C, it is possible to extract the length of an array passed to a function. Also, unlike C, the argument to `main` is compulsory. Of course, the name of the argument is not important: in place of `args` we could have used `xyz` or any other valid identifier.

Here then, is a Java equivalent of the canonical *Hello world* program.

```
class helloworld{
  public static void main(String[] args){
    System.out.println("Hello world!");
  }
}
```

Here `println` is a static method in the class `System.out` which takes a string as argument and prints it out with a trailing newline character.

As noted earlier, we have to save this class in a file with the same name and the extension `.java`; that is, `helloworld.java`.

2.2 Compiling and running Java programs

The java compiler is invoked using `javac`. Though our Java program may consist of several classes, it is sufficient to invoke `javac` on the class containing the static method `main` where we want the execution to begin. In the simple example above, we would write

```
javac helloworld.java
```

Note that we have to supply the full filename, with the extension `.java`, to `javac`.

The Java compiler will automatically compile all additional classes that are invoked (transitively) by the current class being compiled. Each compiled class is stored in a file with the extension `.java` replaced by `.class`. In our simple example, we get a single new file called `helloworld.class`.

To get the full effect, try splitting the example into two classes (in two separate files), as follows, and see what happens when you compile `helloworld.java`:

```
class helloworld{
  public static void main(String[] args){
    auxclass.printit("Hello world!");
  }
}

class auxclass{
  public static void printit(String s){
    System.out.println(s);
  }
}
```

The Java compiler does not produce executable machine code for the underlying architecture, unlike, say, gcc. Instead, Java programs execute on a uniform underlying *virtual machine* called the Java Virtual Machine (JVM). The compiler produces JVM machine code, also called JVM bytecode.

To run the program, we need an interpreter—that is, an implementation of the JVM. This is invoked using the command `java`. We write

```
java helloworld
```

to load the compiled file `helloworld.class` into the JVM interpreter and execute it. (Note that we should *not* give the extension `.class` in this case, unlike when we invoked the compiler `javac`!)

The fact that Java programs run under a uniform "artificial" environment (the JVM) on all architectures and operating systems makes it possible for Java to achieve a higher degree of standardization than most other programming languages—for instance, there is no ambiguity about the size of an `int`, unlike C. Moreover, since the JVM is in complete control, more exhaustive error reporting and handling is possible for runtime errors such as array bound overflows. Finally, dynamic storage management is significantly simplified by the incorporation of an automatic garbage collector in the JVM to reclaim freed memory, removing the peril of "memory leaks". Of course, there is a performance penalty, but that is another story.

2.3 Basic Java syntax

2.3.1 Scalar variables

The basic variable types are similar to C. The basic types are:

`int` (4 bytes), `long` (8 bytes), `short` (2 bytes)
`float` (4 bytes), `double` (8 bytes)
`char` (2 bytes)
`boolean`

Some points to note.

1. The size of each variable is unambiguously fixed (since all Java programs run on a uniform Java Virtual Machine, independent of the local architecture).

2. The `char` type is 2 bytes, not 1 byte. Java uses an encoding called Unicode which allows all international character sets to be written as `char`. Unicode is a conservative extension of ASCII, so if the first byte is 0, the second byte corresponds to the normal ASCII code. We need not worry about Unicode. As far as we are concerned, what is important is that character constants are enclosed in single quotes, as usual—e.g.,

```
char c;
c = 'a';
c = 'X';
if (c != '}') {...}
```

3. There is an explicit boolean type, unlike C. The two constants of this type are written `true` and `false`. We can write code like:

```
boolean b, c = false;
b = true;
b = (i == 7);
```

   Another point is that a typo like `if (x = 7) ...` will be caught as syntactically illegal since `x = 7` returns `7`, not the boolean value `true`.

2.3.2 Expressions

Expressions are essentially the same as C, including boolean expressions. As in C, an assignment of the form `x = expr` returns a value, namely the final value of the expression `expr` that is assigned to the variable `x`. Hence, like C, for checking equality, use `==` not `=`, but (as noted above), stupidities like `if (x = 7)....` will be caught by the compiler.

2.3.3 Compound statements

Conditional execution

`if (condition) ... else ...` , same as in C

Loops

```
while (condition) { ... }
do { ... } while (condition)
for (i = 0; i < n; i++) { ... }
```

Goto

There is no `goto` in Java. However, the language has labelled `break` statements that allow several levels of nested loops to be terminated in one go. For example:

```
outer_loop:                 // this is a loop label
 for (i = 0; i < n; i++){
   for (j = 0; j < n; j++){
     if (a[i][j] == k){
        break outer_loop;  // exits the outer for loop
     }
   }
 }
```

Comments

Comments can be C-style, enclosed in `/* ... */` (but remember that these are not allowed to be nested, as in C).

Another possibility is a single line comment starting with `//` — the rest of the line after `//` is treated as a comment and ignored. See examples above in the "labelled break" explanation.

Strings

Strings in Java are handled using the built-in class `String`. String variables are declared as follows:

```
String s,t;
```

String constants are enclosed in double quotes, as usual. Thus, one can write:

```
String s = "Hello", t = "world";
```

Strings are *not* arrays of characters, so one cannot refer to `s[i]`, `t[j]` etc. To change the contents of string `s` from `"Hello"` to `"Help!"` we cannot just say:

```
s[3] = 'p'; s[4] = '!';
```

What we can do is invoke the method substring in the String class and say something like:

```
s = s.substring(0,3) + "p!";
```

Here, `s.substring(0,3)` returns the substring of length 3 starting at position 0. The operator `+` is overloaded to mean concatenation for strings.

String objects are *immutable*—if a string changes, you get a new `String` object. Even though, in principle, the new value of `s` fits in the same amount of space as the old one, Java creates a new object with the contents `"Help!"` and `s` points to this new object henceforth.

What happens to the old object? It is automatically returned to the system: Java does runtime "garbage collection" so we do not have to explicitly return dynamically allocated storage using something like `free(..)`, unlike in C.

Remember that an expression like

```
s = "A new piece of text";
```

is an abbreviation for something like

```
s = new String("A new piece of text");
```

The string passed as a parameter to `new` is converted into the appropriate internal representation by the constructor for `String` class. The point that needs to be noted is that though `String` is a built in class like any other and has its own methods etc, there are some short cuts built-in to Java, such as:

- Creating a new `String` object implicitly using an assignment statement.
- Overloading `+` to mean concatenation, so we can have expressions of type `String`.

The length of a string `s` can be obtained by calling the method `s.length()`. Note that we don't know (and don't care) how strings are actually represented in the `String` class—for instance, we don't have the C headache of remembering that we always need to put a `'\0'` at the end of a string to make it compatible with functions that operate on strings.

For more information on what methods are available in `String`, look at the documentation for Java, available on the system.

Arrays

Like strings, arrays are objects in Java. A typical declaration looks like this:

```
int[] a;
a = new int[100];
```

This can also be done as follows (a bit more C like);

```
int a[];
a = new int[100];
```

However, arguably the first definition is clearer in that it clearly declares `a` to be of type `int[]` and not `int`.

These two lines can be combined as:

```
int[] a = new int[100];
```

Observe however, that declaring an array is fundamentally a two step process — you declare that `a` is an array (without specifying its size) and then create the array with a fixed size at run time. Of course, you can abandon the current array and create a new one with a fresh call to `new`, as follows. (This example also shows that you can genuinely assign the size of the array at run time.)

```
public class arraycheck{

  public static void main(String[] argv){

    int[] a;
    int i;
    int n;

    n = 10;

    a = new int[n];

    for (i = 0; i < n; i++){
        a[i] = i;
    }

    n = 20;

    a = new int[n];
```

```
      for (i = 0; i < n; i++){
          a[i] = -i;
      }

      for (i = 0; i < n; i++){
          System.out.print(i);
          System.out.print(":");
          System.out.println(a[i]);
      }
    }
  }
```

A very useful feature of arrays in Java is that the length of an array is always available: the length of an array a is given by a.length. A method such as a sort program that manipulates an array does not require an additional argument stating the length of the array to be manipulated, unlike in C.

2.3.4 Final note

Now we know exactly what type the argument to main() is in the following:

```
  class helloworld{
    public static void main(String[] args){
      System.out.println("Hello world!");
    }
  }
```

Here, args is an array of String. We don't need the equivalent of argc in C because we can extract it via args.length.

Chapter 3

Subclasses and inheritance

3.1 Subclasses and superclasses

Consider the following definition:

```
class Employee
{
  private String name;
  private double salary;

  // Constructors
  public Employee(String n, double s){
     name = n; salary = s;
  }

  public Employee(String n){
     this(n,500.00);
  }

  // "mutator" methods
  public boolean setName(String s){ ... }

  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }

  public double getSalary(){ ... }


  // other methods
```

```
   double bonus(float percent){
      return (percent/100.0)*salary;
   }
}
```

Suppose we want to define a class of Managers, who are like Employees except that they have, in addition, a secretary. We can "re-use" the Employee class as follows:

```
class Manager extends Employee
{
        private String secretary;

        public boolean setSecretary(name s){ ... }

        public String getSecretary(){ ... }

}
```

Objects of type Manager implicitly inherit instance variables and methods from Employee. Thus, every Manager object has a name and salary in addition to a secretary.

In OO terminology, Manager is referred to as the *subclass* while Employee is referred to as the *superclass*. In general, a subclass specializes the superclass. The terminology makes sense set-theoretically: every Manager is also an Employee so the set of Managers is a subset of the set of Employees.

The class Manager cannot see the private fields of Employee! This is to prevent the data encapsulation from being broken—often the programmer who creates a subclass is not the one who wrote the parent class. A constructor for Manager *must* call Employee constructor to set name and salary. In a hierarchy of classes, this transfers the responsibility for construction of higher levels to the parent class, which inductively "knows what to do". We use the word super to call the parent constructor. Thus, we would write:

```
// Constructors
public Manager(String n, double s, String sn){
   super(n,s);      // "super" calls the Employee constructor
   secretary = sn;
}
```

Manager can override a method from Employee. For instance, Manager may redefine bonus as follows—note the use of super to unambiguously use the superclass's bonus in this definition.

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

In general, the subclass has greater functionality (more fields and methods) than the superclass. So, casting "up" the hierarchy is permitted. For example,

```
 Employee e = new Manager ( ....)
```

A `Manager` object can do everything that an `Employee` object can. The converse is not possible. What would the reference to `getSecretary()` mean in the following code?

```
Manager m = new Employee(...);
System.out.println(m.getSecretary());
```

3.2 Inheritance polymorphism

The static type of the variable determines what methods are accessible. If we write

```
Employee e = new Manager ( ....)
```

we are *not* permitted to write

```
System.out.println(e.getSecretary());
```

even though, at run time, the call to `getSecretary()` is not a problem.

But, we still have to answer the following. What does

```
e.bonus(p)
```

mean?

There are two possibilities:

1. **Static:** `e` is an `Employee`, so take the definition of `bonus` from class `Employee`.
2. **Dynamic** `e` is declared to be an `Employee` but is actually a `Manager` at run time, so take the definition of `bonus` from `Manager`.

The dynamic definition is more useful than the static one—it permits the following type of "generic" program.

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager e = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0);
}
```

Here, emparray[1] will correctly use bonus as defined in Manager.

This ability to choose the appropriate method at run time, based on the identity of the object, is called *run time polymorphism* or *inheritance polymorphism*.

This is in contrast to *overloading based polymorphism* such as the multiplicity of sort methods in the class Arrays, described earlier.

Getting back to the code above, is emparray[1].setSecretary(s) allowed? After all, we know that it is a Manager, even though it is declared Employee.

As mentioned earlier, if a variable is declared to be of type Employee, only methods that are accessible in the class Employee may be used.

However, if we are confident that at run time the object is really a Manager, we may ask for a "type upgrade", or a *cast*. Casting is done using C-like syntax.

```
((Manager) emparray[1]).setSecretary(s)
```

At run time, if emparray[1] is *not* a Manager, the cast will fail (rather than call an absurd method) and the program will crash. Casting can only be done between compatible types—that is, from a class to one of its descendants.

We can check whether emparray[1] is a Manager at run time using the predicate instanceof. To be safe, we can write:

```
if (emparray[1] instanceof Manager){
  ((Manager) emparray[1]).setSecretary(s);
}
```

This is a trivial example of a property called *reflection*. Reflection is used in the philosophical sense to refer to a language that can "think about itself". Java is rich in reflective capabilities. We will discuss this again later.

3.3 Multiple inheritance

Is it possible for a class to extend more than one class. For instance, could we have the following?

```
class C1{
  ...
}

class C2{
  ...
}

class C3 extends C1, C2{
  ...
}
```

The problem is that what C3 inherits from C1 might conflict with what it inherits from C2. Suppose we have a method f(..) in both C1 and C2 as follows:

```
class C1{
  ...
  public int f(int i){ ... }
  ..
}

class C2{
  ...
  public int f(int i){ ... }
  ..
}

class C3 extends C1, C2{
  ...     // no definition of "public int f(int i)" in C3
}
```

Now, if we create an object of type C3 and invoke f(..), it is not clear whether we should use the definition of f(..) from C1 or C2.

One possibility is to say that C3 can extend C1 and C2 provided it explicitly redefines all conflicting methods. Another is to say that C3 can extend C1 and C2 provided there are no conflicting methods. The latter option is available in C++.

3.4 The Java class hierarchy

Java rules out multiple inheritance. In Java, the class hierarchy is tree like.

In fact, not only is the hierarchy tree-like, Java provides a universal superclass called Object that is defined to be the root of the entire class hierarchy. Every class that is defined in a Java program implicitly extends the class Object.

The class Object defines some useful methods that are automatically inherited by all classes. These include the following:

```
boolean equals(Object o)  // defaults to pointer equality

String toString()         // converts the values of the
                          // instance variable to String
```

The built-in method equals checks whether two variables actually point to the same physical object in memory. This is rather strong and most classes will redefine this to say that two objects are the same if their instance variables are the same.

The `toString()` method is useful because Java implicitly invokes it if an object is used as part of a string concatenation expression. To print the contents of an object `o`, it is sufficient to write:

```
System.out.println(o+"");
```

Here, `o+""` is implicitly converted to `o.toString()+""`.

Recall that a variable of a higher type can be assigned an object of a lower type (e.g., `Employee e = new Manager(...)`) This compatibility also holds for arguments to methods. A method that expects an `Employee` will also accept a `Manager`.

We can write some "generic" programs, as follows:

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length(); i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

As mentioned above, we will probably redefine methods like `equals` to be less restrictive than just referring to pointer equality. However, we should be careful of the types we use. Suppose we enhance our class `Date` to say:

```
boolean equals(Date d){
  return ((this.day == d.day) &&
          (this.month == d.month) &&
          (this.year == d.year));
}
```

Before proceeding, we note a couple of facts about this method. Recall that `day`, `month` and `year` are private fields of class `Date`. Nevertheless, we are able to refer to these fields explictly for the incoming `Date` object `d`. This is a general feature—any `Date` object can directly read the private variables of *any other* `Date` object. Without this facility, it would be tedious, or even impossible, to write such an `equals(..)` method.

Second, note the use of `this` to unambiguously refer to the object within which `equals` is being invoked. The word `this` can be omitted. If we write, instead,

```
boolean equals(Date d){
  return ((day == d.day) &&
          (month == d.month) &&
          (year == d.year));
}
```

then, automatically, the unqualified fields day, month and year refer to the fields in the current object. However, we often use this to remove ambiguity.

Now, what happens when we call our generic find with find(datearr,d), where datearr and d are declared as Date[] datearr and Date d?

When we call datearr[i].equals(d) in the generic find, we search for a method with signature

```
boolean equals(Object o)
```

because, within find, the parameter to find is an Object. This does *not* match the revised method equals in Date, so we end up testing pointer equality after all!

To fix this, we have to rewrite the equals method in Date as:

```
boolean equals(Object d){
  if (d instanceof Date){
    return ((this.day == d.day) &&
            (this.month == d.month) &&
            (this.year == d.year));
  }
  return(false);
}
```

Notice that this is another use of the predicate instanceof to examine the actual class of d at runtime.

In general, the method used is the closest match. If Employee defines

```
boolean equals(Employee e)
```

and Manager extends Employee but does not redefine equals, then in the following code

```
Manager m1 = new Manager(...);
Manager m2 = new Manager(...);
...
if (m1.equals(m2)){ ... }
```

the call m1.equals(m2) refers to the definition in Employee because the call to a non existent method

```
boolean equals(Manager m)
```

is compatible with both

```
boolean equals(Employee e) in Employee
```

and

```
boolean equals(Object o) in Object.
```

However, the definition in Employee is a “closer” match than the one in Object.

3.5 Subtyping vs inheritance

In the object-oriented framework, inheritance is usually presented as a feature that goes hand in hand with subtyping when one organizes abstract datatypes in a hierarchy of classes. However, the two are orthogonal ideas.

- *Subtyping* refers to compatibility of interfaces. A type `B` is a subtype of `A` if every function that can be invoked on an object of type `A` can also be invoked on an object of type `B`.

- *Inheritance* refers to reuse of implementations. A type `B` inherits from another type `A` if some functions for `B` are written in terms of functions of `A`.

In the object-oriented framework, if `B` is a subclass of `A` then it is clear that `B` is a subtype of `A`. Very often, `B` also inherits from `A`—for example, recall the way the implementation of `bonus()` for `Manager` called on the function `bonus()` defined in `Employee`.

However, subtyping and inheritance need not go hand in hand. Consider the data structure *deque*, a double-ended queue. A deque supports insertion and deletion at both ends, so it has four functions `insert-front`, `delete-front`, `insert-rear` and `delete-rear`. If we use just `insert-rear` and `delete-front` we get a normal queue. On the other hand, if we use just `insert-front` and `delete-front`, we get a stack. In other words, we can implement queues and stacks in terms of deques, so as datatypes, `Stack` and `Queue` inherit from `Deque`. On the other hand, neither `Stack` not `Queue` are subtypes of `Deque` since they do not support all the functions provided by `Deque`. In fact, in this case, `Deque` is a subtype of both `Stack` and `Queue`!

In general, therefore, subtyping and inheritance are orthogonal concepts. Since inheritance involves reuse of implementations, we could have an inheritance relationship between classes that are incomparable in the subtype relationship. In a more accurate sense, this is what holds between `Stack`, `Queue` and `Deque`. The type `Stack` supports functions `push` and `pop` that are implemented in terms of the `Deque` functions `insert-front` and `delete-front`. Similarly, `Queue` supporst functions `insert` and `delete` that are implemented in terms of the `Deque` functions `insert-rear` and `delete-front`. The function names are different, so none of these types is comparable in the subtype relation, but `Stack` and `Queue` do inherit from `Deque`.

Chapter 4

Abstract classes

4.1 Abstract Classes

We may use a class only as a collective “name” for a group of related classes. For instance, we might have classes such as

```
class Circle{
  private double radius;
   ...
  public double perimeter(){
   ...
  }
   ...
}

class Square{
  private double side;
   ...
  public double perimeter(){
   ...
  }
   ...
}

class Rectangle{
  private double length;
  private double width;
   ...
  public double perimeter(){
   ...
  }
```

```
    ...
  }
```

and combine these all under a class `Shape` so that

```
  class Circle extends Shape{...}
  class Square extends Shape{...}
  class Rectangle extends Shape{...}
```

We don't actually intend to create objects of the parent type `Shape`, but we can use `Shape` to enforce some common properties of the classes that extend `Shape`. For example, we might want to insist that each subclass of `Shape` define a method with the signature

```
  public double perimeter();
```

Though this method can be defined to yield a sensible value in each subclass of `Shape`, there is no reasonable definition that we can provide for the quantity `perimeter()` in an "abstract" `Shape`. We could, of course, create a dummy function in `Shape`, such as

```
  public double perimeter() { return -1.0; }
```

that returns an absurd value.

This way, each subclass of `Shape` will definitely be able to access a method of the required signature, but we want more: we want to insist that the subclass *redefines* this method to a sensible value.

The way to do this is to specify the signature of `perimeter()` in `Shape`, but say it is only a template that has to be implemented in any subclass. We declare a method to be a template by using the word `abstract` as follows:

```
  public abstract double perimeter();
```

If a class contains an abstract method, it makes no sense to create an instance of that class since it is not fully specified. Such a class is itself called abstract. Thus, we have:

```
  abstract class Shape{
    ...
    public abstract double perimeter();
    ...
  }
```

If a class has any abstract methods, it must be abstract, but the converse is not true. An abstract class may have concrete instance variables and methods. For instance, we might allow each `Shape` to have an unique identifier and define this field in common as:

```
abstract class Shape{
  private String identifier;

  public Shape(String s){   // Constructor to set private variable
    identifier = s;         // Call with "super(s)" from subclass
  }

  ...
  public abstract double perimeter();
  ...
}
```

In fact, technically a class may be declared abstract even though it contains *no* abstract components!

Let us call a class concrete if it is not abstract: A concrete class is one that we can instantiate as an object. In order for a subclass to be concrete, it must provide an implementation for all abstract methods that it inherits from above. Otherwise, the subclass also becomes abstract.

Though we cannot create instances (objects) of an abstract class, we can declare variables of this class. We can continue to use an array of the higher class to hold a mixed collection of objects from the lower level, as in the example below:

```
Shape sarr[] = new Shape[3];

Circle c = new Circle(...);   sarr[0] = c;
Square s = new Square(...);   sarr[1] = s;
Rectangle r = new Rectangle(...);  sarr[2] = r;

for (i = 0; i < 2; i++){
  size = sarr[i].perimeter(); // calls the appropriate method
  ...
}
```

4.2 Generic functions

We can also use abstract classes to group together classes that are at different, incomparable points in the class hierarchy but share some common properties. For example, we may want to group together all classes whose objects can be compared using a linear order. Any such class can then serve as a valid input to a sort function, using the comparison method defined within the class whenever the sort function needs to compare two objects.

For instance, we might have a definition such as

```
abstract class Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s, 0 if this == 0, +1 if this > s
}
```

Now, we can sort any array of objects that extend the class `Comparable`:

```
class Sortfunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use a[i].cmp(a[j])
    // (Why is this method declared to be static?)
  }
  ...
}
```

If we want to make the objects of a class sortable using one of the `Sortfunctions`, we write

```
class Myclass extends Comparable{
  double size;   // some quantity that can be used to compare
                 // two objects of this class
  ...
  public int cmp(Comparable s){
    if (s instanceof Myclass){
      // compare this.size and ((Myclass) s).size
      // Note the cast to access s.size
      ...
    }
  }
}
```

4.3 Multiple inheritance and interfaces

What if we want to make a class such as `Circle` (defined earlier) extend `Comparable`? `Circle` already extends `Shape` and Java does not permit multiple inheritance.

Notice, however, that `Comparable` is a special kind of abstract class, one that has *no* concrete components. In Java, this kind of abstract class can be renamed an *interface*—the word interface is derived from its English meaning and refers to the "visible boundary" of a class that implements all the methods specified in the interface.

We can redefine `Comparable` as follows:

```
interface Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s, 0 if this == 0, +1 if this > s
}
```

A class that extends an interface is said to “implement” it:

```
class Myclass implements Comparable{
  ...
  public int cmp(Comparable s) { ... }
  ...
}
```

In Java, a class may extend only one other class, but may implement any number of interfaces. Thus, one can have

```
class Circle extends Shape implements Comparable{
   ...
  public double perimeter(){...}
   ...
  public int cmp(Comparable s){...}
   ...
}
```

As we mentioned earlier, extending multiple classes might create a conflict between concrete definitions in the multiple parent classes. However, since interfaces have no concrete components, no such contradiction can occur if a class implements multiple interfaces. At most, a class may implement two interfaces that both define abstract methods with the same signature. Since the signature does not indicate anything about the “meaning” of the method, any implementation of the method with that signature will be automatically consistent with both its abstract definitions.

Chapter 5

Interfaces: Applications

As we have seen, interfaces can be used to tie together disparate classes that share some common features. For instance, all classes that implement the interface `Comparable` describe objects that can be linearly ordered by some internal parameter and hence sorted.

Notice that the property of being `Comparable` is orthogonal to any other features that the class may possess. Functions that operate on objects that implement the `Comparable` interface, such as the generic sorting functions we described, are not interested in these other features. Indeed, one can make a stronger statement and say that these functions can *only* access the Comparable portion of the objects that are passed to them.

In a more general sense, interfaces provide a limited public view of the overall capabilities of an object. We look at two examples to illustrate this idea.

5.1 Callback functions

As we shall see later, Java permits parallel execution. One function can invoke another function to run in a parallel *thread*. Unlike a conventional function call, where the calling function is suspended till the called function terminates and returns, in this case the calling function proceeds to the next statement after the function invocation, while the invoked function executes in parallel. This parallelism may be actual, such as on a multiprocessor machine, where each thread is assigned to a separate processor, or virtual, such as on a single processor, where the threads are interleaved artibrarily to simulate parallel execution.

Suppose now that we have two classes, `Myclass` and `Timer`, where an object `m` of `Myclass` can create a `Timer` object `t` and invoke a function `t.f()` to run in parallel with `m`. Recall that we have assumed that `m` is not suspended after invoking `t.f()`. We would like a mechanism for `t.f()` to report back to `m` when it has completed its work. To do this, `t` needs to have a reference back to `m`. This can be accomplished by passing the reference when we create `t`: each instance of `Timer` is created by an object and the instance of `Timer` remembers the identity of the object that created it. For instance, we might have:

```
class Myclass{
  ...
  public void some_function(){
    ...
    Timer t = new Timer(this);  // Tell t that this object
                                // created t
    ...
    t.f();                      // Start off t.f() in parallel
    ...
  }
  ...
}
```

We need the following complementary structure in Timer:

```
class Timer implements Runnable{  // Says that Timer can be invoked
                                  // in a parallel thread
  private Object owner;

  public Timer(Object o){         // Constructor
    owner = o;                    // Remember who created me
  }
  ...
  public void f(){
    ...
    o.notify();                   // Tell my creator that I'm done
  }
  ...
}
```

The problem with this definition is that the function o.notify() cannot be invoked as stated without casting o (which is an Object) back to Myclass. Of course, we could fix this by changing the type of owner from Object to Myclass, but then we would have to write a different Timer class for each type of owner.

In order to define a generic Timer class, all we need is that the owner have a method called notify() and that there is a uniform way to cast the owner to a type that can access this method. This can be achieved via an interface, say

```
interface Timerowner{
  public abstract void notify();
}
```

Now, we modify Myclass to implement Timerowner:

```
class Myclass implements Timerowner{
  ...
```

```
    public void some_function(){
      ...
      Timer t = new Timer(this);  // Tell t that this object
                                  // created t
      ...
      t.f();                      // Start off t.f() in parallel
      ...
    }

    public void notify(){ ... }   // Implement the interface
    ...
  }
```

Finally, we modify `Timer` to insist that the owner of a `Timer` object must implement `Timerowner`:

```
  class Timer implements Runnable{  // Can be invoked in a
                                    // parallel thread
    private Timerowner owner;       // Owner must implement Timerowner

    public Timer(Timerowner o){     // Constructor
      owner = o;                    // Remember who created me
    }
    ...
    public void f(){
      ...
      o.notify();                   // Tell my creator that I'm done
    }
    ...
  }
```

Even though the timer `t` gets a reference to its parent object, the parent only “exposes” its `Timerowner` features to `t`.

5.2 Iterators

A linear list is a generic way of storing `Objects` of any type. We might define a class to store generic linear lists:

```
  class Linearlist {
    // Implement list as an array with arbitrary upper bound limit
    private int limit = 100;
    private Object[] data = new Object[limit];
    private int size;  // Current size of the list

    public Linearlist(){  // Constructor
      size = 0;
    }

    public void append(Object o){
      data[size] = o;
      size++;
      if (size > data.length){ // Get more space!
        Object[] tmp = data;   // Save a pointer to current array
        limit *= 2;
        data = new Object[limit]; // New array of double the size
        for (i = 0; i < tmp.length; i++){
          data[i] = tmp[i];
        }
    }
    ...
  }
```

Alternately, we might modify the internal implementation to use a linked list rather than an array.

```
class Node{
  private Object data;
  private Node next;

  public Node(Object o){
    data = o;
    next = null;
  }
}

class Linearlist {

  private Node head;    // Implement list as a linked list
  private int size;

  public Linearlist(){  // Constructor
    size = 0;
  }

  public void append(Object o){
    Node m;

    // Locate last node in list and append a new node containing o
    for (m = head; m.next != null; m = m.next){}
    Node n = new Node(o);
    m.next = n;

    size++;
  }
  ...
}
```

Now, what if we want to externally run through a `Linearlist` from beginning to end? If we had access to the array data we could write

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

Alternatively, if we had access to `head`, we could write

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

However, we have access to neither. In fact, we do not even know which of the two implementations is in use! The `Linearlist` class should add functionality to support such a list traversal. Abstractly we need a way to do the following:

```
Start at the beginning of the list;
while (there is a next element){
  get the next element;
  do something with it
}
```

We can encapsulate this functionality in an interface called `Iterator`:

```
public interface Iterator{
  public abstract boolean has_next();
  public abstract Object get_next();
}
```

Now, we need to implement Iterator in Linearlist. If we make this a part of the main class, there will be a single "pointer" stored within the class corresponding to the temporary variable i (or m) in the loop we wrote earlier to traverse the list. This means that we will not be able to simulate a nested loop such as the following:

```
for (i = 0; i < data.length; i++){
  for (j = 0; j < data.length; j++){
     ... // do something with data[i] and data[j]
  }
}
```

because when we execute the equivalent of the inner loop, we lose the position that we were in in the outer loop.

A better solution is to allow Linearlist to create, on the fly, a new object that implements Iterator and return it outside. This object will be an instance of a nested class, defined within Linearlist. It will have access to private information about the list, even though it is invoked outside! Here is how we do it.

```
class Linearlist {
  // Implement list as an array with arbitrary upper bound limit
  private int limit = 100;
  private Object[] data = new Object[limit];
  private int size;  // Current size of the list

  public Linearlist(){..}  // Constructor

  public void append(Object o){...}

  // A private class to implement Iterator
  private class Iter implements Iterator{
    private int position;

    public Iter(){   // Constructor
      position = 0;  // When an Iterator is created, it points
                     // to the beginning of the list
    }

    public boolean has_next(){
      return (position < data.length - 1);
    }

    public Object get_next(){
      Object o = data[position];
```

```
        position++;
        return o;
      }
    }

    // Export a fresh iterator
    public Iterator get_iterator(){
      Iter it = new Iter();
      return it;
    }
    ...
  }
```

Now, we can traverse the list externally as follows:

```
  Linearlist l = new Linearlist();
  ...
  Object o;
  Iterator i = l.get_iterator()

  while (i.has_next()){
    o = i.get_next();
    ...   // do something with o
  }
  ...
```

What if Linearlist is implemented as a linked list rather than an array? We just change the definition of the class Iter within Linearlist.

```
    private class Iter implements Iterator{
      private Node position;

      public Iter(){   // Constructor
        // Create a dummy node that points to the head of the list
        position = new Node(null);
        position.next = head;
      }

      public boolean has_next(){
        return (position.next != null);
      }

      public Object get_next(){
```

```
      position = position.next;
      Object o = position.data;
      return o;
    }
  }
```

Externally, the loop that we wrote with `get_iterator()`, `has_next()` and `get_next()` works as before.

Once again, we have passed a reference to an object but restricted access to the object using an interface. The `Iter` object created within `Linearlist` can (and must) access private variables within `Linearlist`, but the reference that is passed to this internal object is constrained by the `Iterator` interface, so this capability cannot be exploited externally.

In fact, we can go one step further and even nest the linked list class as a local class within Linearlist to prevent any accidental misuse. A complete implementation of `Linearlist` as a linked list might be as follows.

```
class Linearlist {

  private Node head;       // Implement list as a linked list
  private int size;

  public Linearlist(){...} //Constructor

  public void append(Object o){...}


  private class Node{
    private Object data;
    private Node next;

    public Node(Object o){
      data = o;
      next = null;
    }
  }

  private class Iter implements Iterator{
    private Node position;

    public Iter(){   // Constructor
      // Create a dummy node that points to the head of the list
      position = new Node(null);
      position.next = head;
    }

    public boolean has_next(){
      return (position.next != null);
    }

    public Object get_next(){
      position = position.next;
      Object o = position.data;
      return o;
    }
  }

  // Export a fresh iterator
  public Iterator get_iterator(){
    return new Iter();
  }
  ...
}
```

Notice that when we return an iterator, since we do not need to ever use a name for the iterator locally, we can directly pass the reference generated by new, rather than assigning it temporarily to a locally defined reference (such as `Iter it` in the original version) and returning that local variable.

5.3 Ports and interaction with "state"

An `Iterator` is a simple example of a situation where the interaction with an object is not memoryless. Normally, when an object provides access to a private variable `localdata` through an accessor method `get_localdata()`, each call to `get_localdata()` is independent and the value returned does not depend on how many times, if any, `get_localdata()` has been invoked earlier.

In contrast, the accessor method `get_next()` provided by the `Iterator` interface returns different values depending on the history of previous calls to the same method. Thus, some state information has to be maintained by the object to ensure that the value returned by `get_next()` is accurate.

In general, the OO paradigm does not provide any direct way of specifying such "interactions with state" with an object. The word *port* has been used recently in certain specification methodologies (e.g., UML) to refer to a point of interaction with an object that is accompanied by a state. As we have seen with `Iterators`, ports can be simulated indirectly by exporting objects from within a class that implement a specified interface.

Chapter 6

Java Interlude 2
Parameter passing, cloning, packages

6.1 Parameter passing in Java

In Java, scalar variables (i.e. of type `int`, `long`, `short`, `float`, `double`, `byte`, `char`, `boolean`) are always passed to functions by value, like in C. In other words, the value passed to the function is copied into the local parameter and any change to the local copy has no effect on the original copy.

Unlike C, there is no way to pass the address of a scalar. We cannot write the equivalent of the following code in C:

```
int i,j;
...
swap(&i,&j);
...

void swap(int *p, int *q){
  int tmp;
  tmp = *p;
  *p = *q;
  *q = tmp;
}
```

Objects (i.e., any variable that is not a scalar) are passed by reference, like arrays in C. Like C, the justification is that objects can be large structures and it would be inefficient to copy the structure. Notice, however, that (like in C), we can modify the contents of what a parameter points to, but we cannot modify where it points. If we want to swap the contents of two objects from the same class, we *cannot* write:

```
 class Myclass{
  ...
  public void swap(Myclass p){  // Swap "this" with p
    Myclass tmp;
    tmp = p;
    p = this;
    this = tmp;
  }
}
```

Instead, we must write something like:

```
class Myclass{
  ...
  public void swap(Myclass p){
    Myclass tmp = new Myclass(...);  // Make a new tmp object
    ... // Copy contents of p into tmp
    ... // Copy contents of this into p
    ... // Copy contents of tmp back into this
  }
}
```

What about return values? Suppose we enhance the `Employee` class as follows:

```
class Employee{
  private String name;
  private double salary;
  private Date joindate;

  // Constructor
  public Employee(String n, double s, int d, int m, int y){
     name = n; salary = s;
     joindate = new Date(d,m,y);  //
  }


  ...
  // "accessor" methods
  public Date get_joindate(){ return joindate; }
  ...
}
```

where `Date` is defined as follows:

```
class Date{
  private int day, month, year;
  public Date(int d, int m, int y){
    day = d;
    month = m;
    year = y;
  }

  public void advance(int d){   // Modify day/month/year so that
    ...                         // date advances by d days
  }
  ...
}
```

Now, suppose we use get_joindate() as follows:

```
Employee e = new Employee(...);
Date d = e.get_joindate();
d.advance(100); // e loses 100 days seniority!
```

The problem is that get_joindate() returns a reference to its private copy of Date and this reference can be used to invoke public methods on the Date object, though the Date object is supposed to be a private component of the Employee e.

The solution is to make a copy of the object before returning it.

```
class Employee{
  ...
  public Date get_joindate(){
    Date tmpdate = new Date(...); // Construct tmpdate using
                                  // current values of joindate
    return tmpdate;
  }
  ...
}
```

Here, copying Date is not difficult because Date is itself a "flat" object and can be copied component-wise. If we have to make a copy of Employee, we have to first recursively make a copy of the nested object Date and then copy the scalars salary and name.

6.2 Cloning

The root class Object in Java provides a method

```
Object clone(Object o)
```

that makes a bit-wise copy of its argument. Making a bit-wise copy is fine if the the object being copied consists only of scalars. However, if it has nested objects (like `Date` inside `Employee`), the bitwise copy will copy the reference to the nested object `Date` so the clone will accidentally have a reference to a non-cloned internal component. We should, instead, recursively clone the internal object.

To ensure that a programmer is aware of the perils of using the `clone` method, Java requires that any class that invokes `clone` must implement the interface `Cloneable`. To use clone in `Employee`, we must write

```
class Employee implements Cloneable{
  ...
}
```

(Technically, we also need to redefine clone to be a public method. To understand this, we need to look at accessibilities other than `public` and `private` in Java, which we will do shortly.)

What does the interface `Cloneable` specify? Nothing! It is an empty interface, also called a *marker interface* that merely allows the method `clone()` to check if the class in which it is being invoked has been written by a “thoughtful” programmer who has understood the perils of making a bit-wise copy and (hopefully) taken necessary steps to make it work correctly.

The code for the method `clone` would contain a check something like the following:

```
Object clone(Object o){
  if (o instanceof Cloneable){
     ... // go ahead and clone
  }else{
     ... // complain and quit
  }
}
```

6.3 Packages

So far, we have identified two basic modifiers indicating the range within which a variable or function is accessible: `public` and `private`. As we have said already, public data is visible to all classes and private data is visible only within a class (but to all instances of the same class, so one object of a class can freely examine private variables of another object of the same class).

In all this, we have also frequently written definitions *without* modifiers, such as

```
class LinearList{
   ...
}
```

How visible are such definitions?

Java has a separate mechanism to group togther classes using *packages*. A package, roughly speaking, is a group of classes that are defined in the same directory. To indicate that the class XYZ defined in the file XYZ.java belongs to the package named ABC, we add the line package ABC as the *first* line in XYZ.java. There is a (somewhat arbitrary) restriction on package names: all files in package ABC must live in a directory called ABC.

If a file does not carry a package header, its contents implicitly belong the (unique) anonymous package. All files without a package header in the same directory belong to the same anonymous package.

Definitions without modifiers are visible within the same package. The class definition above is "public" within the package but "private" outside the package. So, in our toy programs, all definitions without modifiers behave like public definitions so long as the definitions lie within the same directory (and thus belong to the anonymous package).

Java packages are arranged in (multiple) directory trees. Java searches for classes using the variable CLASSPATH, similar to the PATH variable used by the Unix shell to search for executable files. Suppose CLASSPATH contains the directory /usr/local and under this, we have a class def that lies in the subdirectory /usr/local/java/xyz/abc. Then, the package name associated with def corresponds to the directory abc and is given by java.xyz.abc while the fully qualified class name of def is java.xyz.abc.def. This reflects the path to reach def relative to the CLASSPATH, with / replaced by ..

The builtin classes in java typically lie in packages of the form java.xyz.abc. To use classes defined in a package we can use an import statement at the top of the file. For instance, if we write

```
import java.math.BigDecimal
```

then the class BigDecimal defined under .../java/math becomes accessible in the current program. More generally, we write

```
import java.math.*
```

to make *all* classes defined in .../java/math accessible to the program. Note here that the * is not recursive, so it does not make available any classes that may lie in subdirectories of .../java/math.

If we do not use the import statement, we can always explicitly select a class by giving its fully qualified name (much like giving the full pathname of a Unix command) and write, for instance,

```
java.math.BigDecimal d = new java.math.BigDecimal();
```

instead of

```
import java.math.*
BigDecimal d = new BigDecimal();
```

Packages are intended to correspond to natural units of software development so it makes sense for some definitions to be fully visible within the "current project" but not outside.

6.4 The protected modifier

In addition to public, private and no modifier (implicitly indicating package visibility), there is another modifier called protected, indicating visibility within the inheritance subtree rooted at a class—that is, the collection of all classes that directly or indirectly extend the current class.

While the usefulness of protected mode instance variables is doubtful, it does make some sense to have protected mode functions, that can be used by subclasses that can be "trusted" to know what these functions are doing, but not by arbitrary classes outside the subtree.

It is allowed for a subclass to promote a function from protected to public while redefining it. Note that this is a specific exception to the general rule that says that a subclass is not allowed to change the visibility of a function when redefining it. (It is also not permissible for a subclass to change the return type of a function when redefining it.)

One concrete example where this is used is in the function clone() defined in Object. As we have already noted, clone() makes a bitwise copy, which is not necessarily the correct way to clone an object, so the clone() method checks that the calling class implements the marker interface Cloneable() to verify that the programmer actually intends that clone() be used.

As a further precaution against accidental misuse, clone() is declared to be protected in Object. This means that by default, even after implementing the Cloneable interface, only subclasses can invoke clone() on objects of a given class. To make an object fully cloneable, the class must redefine clone() with the modifier public.

Part II

Exception handling

Chapter 7

Exception handing

In programming language parlance, an *exception* is an unexpected event that disrupts the normal execution of a program. Exactly what constitutes an exception can be a subjective decision. For instance, an illegal operation such as division by zero or accessing an array element beyond the bound of the array is almost surely an exception. How about reaching the end of the file when reading a file? This can be viewed as an interruption of the "normal" operation of reading a file. Java takes the stand that reaching the end of file is an exceptional situation. Most other programming languages would agree that though this is a special situation, it will generally occur at the end of every read loop and so can be dealt with in the normal program flow (such as checking that the character read is `EOF`).

The key problem is to deal with exceptions in a sensible way. The aim is for the program to be able to recover and take corrective action if possible and abort only if there is no other option left. Further, it should be possible to report sensible information about the nature of the exception for diagnostic purposes.

In C, a very rudimentary form of exception reporting is done by examining the return value of a function such as `main()`. Often, a return value of `int` encodes the result of the computation of the function—minimally, a return value 0 may indicate no error while a non-zero value might indicate an error. The type of error might be indicated by the actual non-zero return value—returning a value 1 might indicate malformed input while returning a value 2 might indicate the inability to open a file that was need for the program to run.

7.1 Java's `try-catch` blocks

Java treats an exception as an object with structure. When an error happens, an appropriate exception object is *thrown* back to the program that generated the error. The instance variables of this object contain information about the nature of the exception. At the very least, there is an error message stored in the object as a string.

If the program is written properly, it can *catch* the exception object, analyze its contents to determine the cause of the error and then take corrective action if necessary. This is known as *handling* the exception.

In Java, all exception objects are subclasses of a class called Throwable. There are two immediate subclasses of Throwable, called Error and Exception. The class Error covers exceptional situations that arise because of circumstances beyond the program's control—for instance, the kind of error that occurs if a file that needs to be read cannot be found on the disk. The other subclass, Exception, has one subclass RunTimeException, that captures standard runtime computational errors such as divide-by-zero and out-of-bounds-access of an array.

Java supports a try-catch structure to *catch* exceptions:

```
try{
     ... // Code that might generate error
     ...
   }
catch (ExceptionType1 e1){...} // Analyze contents of e1
                               // Corrective code for ExceptionType1

catch (ExceptionType2 e2){...} // Analyze contents of e1
                               // Corrective code for ExceptionType2
```

If an error occurs in the try block, the exception object generated is passed to the (multiple) catch statements accompanying the try block. The type of the exception object is checked sequentially against each catch statement till it matches. In the example above, if the first catch statement were to be rewritten as

```
catch (Throwable e1){...}
```

all exception objects would match Throwable and be handled by this piece of code, ignoring the rest. One should, in general, arrange the catch statements so that the earlier statements deal with *more* specific exception types than later catch statements.

The sequence of flow when an exception occurs is as follows:

- Abort the code within the try block at the point where the exception occurs.
- Transfer control to the first catch block that matches the thrown exception object.
- If no such block is found, abort and pass on the exception object to the calling class (this allows exceptions to be propagated up to a level that can handle them).
- If some catch block does match, execute this block and then proceed to the next new statement after the catch.

7.2 Cleaning up with `finally`

This does not provide a mechanism to do any cleaning up of system resources (e.g., closing open files) when an exception occurs. The solution is to add a block labelled `finally`, as follows.

```
try{
     ... // Code that might generate error
     ...
   }
catch (ExceptionType1 e1){...} // Corrective code for ExceptionType1

catch (ExceptionType2 e2){...} // Corrective code for ExceptionType2

finally{
     ... // Always executed, whether try terminates normally or
     ... // exceptionally.  Use for cleanup statements
}
```

The semantics of `finally` is that it is *always* invoked, whether or not `try` terminates normally. If the `try` block does not raise an exception, after the `try` block control passes to the `finally` block and then to the next statement. If an exception does occur, the `finally` block is executed after the appropriate `catch` block. If no `catch` block applies, the `finally` block is executed before aborting the function and propagating the error back to the caller.

7.3 Customized exceptions

In addition to catching exceptions, Java allows you to generate your own exceptions using the `throw` statement. The default `Exception` objects can store a string, indicating an error message. In addition, you may add more structure in a user defined subclass of `Exception`. If you do not want class `LinearList` to have negative entries you might write:

```
class NegativeException extends Exception{

  private int error_value;      // Stores negative value that
                                // generated the exception

  public NegativeException(String message, int i){ // Constructor
    super(message);  // Appeal to superclass constructor for message
    error_value = i;
  }

  public int report_error_value(){
    return error_value;
  }
}
```

Now, inside `LinearList` you might write:

```
class LinearList{
  ...
  public add(int i){
    ...
    if (i < 0){
      throw new NegativeException("Negative input",i);
    }
    ...
  }

}
```

Notice that we directly passed the reference returned by new rather than creating a temporary local variable of this type as follows:

```
  public add(int i){
    NegativeException ne;
    ...
    if (i < 0){
      ne =  new NegativeException("Negative input",i);
      throw ne;
    }
    ...
  }
```

When another program uses `add(i)` in a `LinearList`, it should be aware of the possibility that a `NegativeException` may be thrown. Java insists that this exception throwing capability be “advertised” in the function definition, as follows:

```
class LinearList{
  ...
  public add(int i) throws NegativeException{
   ...
  }
  ...
}
```

Actually, Java relaxes the rule about advertising exceptions so that a program can throw a built in exception of type `Error` or `RunTimeException` without advertising it.

We can now use `add(i)` externally as:

```
LinearList l = new LinearList();
...
try{
  ...
  l.add(i);
  ...
}
catch (NegativeException ne){
  System.out.print("Negative input supplied was ");
  System.out.print(ne.report_error_value);
}
...
```

As mentioned earlier, Java takes a rather extreme stand in classifying “non-normal” execution as an exception. Reaching the end of file throws an `IOException`, so the normal structure of a read loop from a file in Java would be:

```
try{
  while(...){
    ...            // read from a file
  }
}
catch (IOException e){
   if (e ... EOF) { ...}  // if e indicates EOF
   ...
}
```

7.4 A “cute” fact about `finally`

Normally, we believe the following two pieces of code to be equivalent:

Iteration using for

```
for (i = 0; i < n; i++){
  statement 1;
  statement 2;
  ...
  statement k;
}
```

Iteration using while

```
i = 0;
while (i < n){
  statement 1;
  statement 2;
  ...
  statement k;
  i++;
  }
```

However, observe what happens if one of the internal statements is `continue` (or, more precisely in Java, `continue l` where `l` is a loop label).

```
l:
for (i = 0; i < n; i++){
  statement 1;
  statement 2;
  ...
  continue l;
  statement k;
}
```

```
l:
i = 0;
while (i < n){
  statement 1;
  statement 2;
  ...
  continue l
  statement k;
  i++;
}
```

The statement `continue l` causes the loop `l` to start its next iteration without executing any of the statements that follow it in the loop. In the `for` loop, this still results in the increment `i++` being executed since it is part of the loop definition and not a statement "inside" the loop. In the `while` loop, on the other hand, the increment `i++` is skipped along with `statement k` so that the loop restarts with the *same* value of `iq`.

To fix this, we should use the `try-finally` construction (without a `catch` clause) as follows:

```
l:
i = 0;
while (i < n){
  try{
    statement 1;
    statement 2;
    ...
    continue l
    statement k;
  }
  finally{
    i++;
  }
}
```

Now, the `continue l` causes the `try` block to terminate prematurely, but the `finally` block is still executed before reentering the loop, so this version does preserve the semantics of the for.

Part III

Event driven programming

Chapter 8

Event driven programming

8.1 Programming graphical user interfaces

When programming modern graphical user interfaces, we have to deal with an important new variety of programming, normally called event driven programming. The operating system captures external "events" such as keystrokes and mouse movements and passes them on to the program we write, which must then react sensibly. For instance, if the mouse is positioned over a button and then clicked, we must activate the function indicated by the button.

At a low level, the minimum support that we need is the ability to run a main routine in parallel with a secondary routine that responds to these events. For instance, if the program is a web browser, then the main routine will display web pages. Whenever a hyperlink is selected, the act of displaying the current page is interrupted and the browser initiates a connection to a new page.

The next question is the form in which the events are passed. At the most basic level, the events of interest are of the form "key `'a'` was pressed", "the mouse moved to position (x, y)" or "the left button of the mouse was pressed". If the program only gets such information from the underlying system, then the programmer has to do a lot of work to keep track of the current position of all the graphical objects being displayed so that the events can be correlated to the position of these objects.

Consider, for instance, how the program would figure out that a button labelled `OK` has been selected by the user. First, the program has to remember the current location of the button on the screen. Next, the program has to track all "mouse move" events to know where the mouse is at any point. Suppose the program now gets an event of the form "mouse click", and the currently recorded position of the mouse is (x, y). If (x, y) lies within the boundary defined by the button labelled `OK`, then the user has actually clicked on the button `OK` and appropriate action must be taken. On the other hand, if (x, y) does not lie within the current boundaries of the `OK` button, then some other button or graphical component has been selected and we then have to figure out which component this is. Or, yet again, it may be that this mouse click is outside the scope of all the windows being displayed by the current program and can hence be ignored.

Needless to say, programming graphical displays at this low level is very tedious and error-prone. We need a higher level of support from the run-time environment of the programming language. The run-time environment should interact with the operating system, receive low level events such as keystrokes and mouse movements and *automatically* resolve these into high level events indicating when a component such as a button has been pressed. Thus, the programmer does not have to keep track of where a graphical component is or which component is affected by a particular low level event: implicitly, whenever a graphical component is selected, it gets a signal and can invoke a function that takes appropriate action.

This support is available in Java through the Swing package, which in turn relies on a lower level part of Java called AWT (Abstract Windowing Toolkit). In Swing, we can directly define graphic objects such as buttons, checkboxes, dialogue boxes, pulldown menus . . . and specify the size, colour, label . . . of these components. Each of these components is defined as a builtin class in Swing. In addition, each component is capable of causing some high level events. For instance, a button can be pressed, or an item can be selected in a pulldown menu. When an event occurs, it is passed to a prespecified function.

How do we correlate the objects that generate events to those which contain the functions that respond to these events? Each component that generates events is associated with a unique collection of functions that its events invoke. This collection is specified as an interface. Any class that implements this interface is qualified to be a *listener* for the events generated by this type of component. The component is then passed a reference to the listener object so that it knows which object is to be notified when it generates an event.

For instance, suppose we have a class `Button` that generates a single type of event, corresponding to the button being pushed. When a button is pushed, we are told that a function called `buttonpush(..)` will be invoked in the object listening to the button push. We handle this as follows:

```
interface ButtonListener{
  public abstract void buttonpush(...);
}

class MyClass implements ButtonListener{
  ...
  public void buttonpush(...){
     ...      // what to do when a button is pushed
  }
  ...
}

...

Button b = new Button();
```

```
MyClass m = new MyClass();
b.add_listener(m);   // Tell b to notify m when it is pushed
```

The important point is that we need not do anything beyond this. Having set up the association between the Button b and the ButtonListener m, whenever b is pressed (wherever it might be on the screen), the function m.buttonpush(...) is *automatically* invoked by the run-time system.

Why does buttonpush(...) need arguments? The event generated by a Button has structure, just like, say, an exception or any other signal. This structure includes information about the source of the event (a ButtonListener may listen to multiple buttons) and other useful data.

In a sense, the Timer example that we saw in Chapter 5.1 fits in this paradigm. Recall that we wanted to start off a Timer object in parallel with the current object and have it report back when it finished. The act of finishing is an event and the function it triggers is the one we called notify(). In the example, the class that created the Timer passed a pointer to itself so that the Timer object could notify it. We could instead have passed a reference to *any* object that implements the interface Timerowner. Thus, the object we pass to Timer is a listener that listens to the "event" generated by the Timer reaching the end of the function f().

The relationship between event generators and event listeners in Java is very flexible. Multiple generators can report to the same listener. This is quite natural—for instance, if we display a window with three buttons, each of which describes some function to be performed on the content of the window, it makes sense for the window to listen to all three buttons and take appropriate action depending on which button is pressed.

More interestingly, the same event can be reported to multiple listeners. This is called multicasting. A typical example is when you want to close multiple windows with a single mouse click. For instance, suppose we have opened multiple windows in a browser. When we exit the browser (one click) all these windows are closed.

This flexibility also means that the connection between event generators and event listeners has to be set up explicitly. If no listener is notified to the event generator, the events that it generates are "lost". In some languages, each component (such as a Button) is automatically associated with a fixed listener object. The advantage is that it is almost automatic that all events are listened to and not lost. However, the disadvantage is a loss of flexibility in the ways we can interconnect event generators and event listeners.

8.2 Some examples of event driven programming in Java

8.2.1 A button

A button is a graphical object that can be clicked. There is only one thing that a user can do with a button—click it. Thus, when a button is clicked, an appropriate listener has to be informed that it was clicked, along with the identity of the button. The built-in button class in Swing is called `JButton`. The interface `ActionListener` describes classes that can listen to button clicks. When a button is clicked, the method `actionPerformed(...)` is invoked in each `ActionListener` that is associated with this button.

Thus, to construct a button in Java, we need to declare a `JButton` and provide a class that listens to it. At the minimum we need the following.

```
class MyButtons{
  private JButton b;

  public MyButtons(ActionListener a){
     b = new JButton("MyButton");  // Set the label on the button
     b.addActionListener(a);       // Associate an listener
  }
}

class MyListener implements ActionListenerP
 public void actionPerformed(ActionEvent evt){
     ...                           // What to do when a button
     ...                           // is pressed
 }

class XYZ{
  MyListener l = new MyListener(); // Create an ActionListener l
  MyButtons m = new MyButtons(l);  // Create a button m, listened to by l
```

Unfortunately, we have to do a lot more to actually display a button. First, we have to generate a “container” to hold the button. Usually, this is done using an object called a *panel*—the corresponding class in Swing is `JPanel`. Here is how to define a panel that contains a single button. The enclosing panel listens to the button click. The `import` statements at the top include the appropriate packages from AWT and Swing.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonPanel extends JPanel implements ActionListener{

    private JButton redButton;

    public ButtonPanel(){
        redButton = new JButton("Red");      // Create the button
        redButton.addActionListener(this);  // Make the panel listen
```

```
                                          //   to the button
        add(redButton);                   // Embed the button in the
                                          //   panel
    }

    public void actionPerformed(ActionEvent evt){
        Color color = Color.red;          // Set the background colour
        setBackground(color);             //   to red when the button
        repaint();                        //   is clicked
    }

}
```

In Java, a panel cannot be displayed directly. We have to embed the panel into a *frame*—in Swing, the class we need is `JFrame`. A `JFrame` can generate seven different types of events that are handled by objects that implement `WindowListener`. These events correspond to actions such as iconizing a window, making a window occupy the full screen, killing the window, ... Each of these actions invokes a different function in `WindowListener`. It is important to note that this classification of the seven different types of actions is done automatically by Swing—we do not need to write an elaborate case-analysis in a `WindowListener` to decipher the nature of the event.

Here is a simple frame that includes a `ButtonPanel`. The class `ButtonFrame` acts as its own `WindowListener`. For six of the seven functions, we have provided just a stub. The only window event that `ButtonFrame` handles is the one that kills the window. If the window is killed, the corresponding Java program exits. If we do not write something appropriate for the method `windowClosing(WindowEvent e)`, when the user kills the window the window manager will delete the frame but the underlying Java program will continue to run "invisibly"!

The only other complication in this example is the use of `ContentPane`. A `JFrame` is a "complex" object, made up of different conceptual "layers". One of these layers is called the `ContentPane`. When we add objects to the `JFrame`, we actually add them to the `ContentPane`. Thus, to add a `ButtonPanel` to a `ButtonFrame`, we first get access to the `ContentPane` of the `ButtonFrame` and add the `ButtonPanel` to the `ContentPane`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonFrame extends JFrame  implements WindowListener {
    Private Container contentPane;

    public ButtonFrame(){
        setTitle("ButtonTest");
        setSize(300, 200);
        addWindowListener(this);              // ButtonFrame is its
                                              //   own listener
        contentPane = this.getContentPane();  // ButtonPanel is added
        contentPane.add(new ButtonPanel());   //   to the contentPane
    }

    // Seven methods required for implementing WindowListener
    // Six out of seven are dummies (stubs)

    public void windowClosing(WindowEvent e){  // Exit when window
        System.exit(0);                        //    is killed
    }

    public void windowActivated(WindowEvent e){}

    public void windowClosed(WindowEvent e){}

    public void windowDeactivated(WindowEvent e){}

    public void windowDeiconified(WindowEvent e){}

    public void windowIconified(WindowEvent e){}

    public void windowOpened(WindowEvent e){}

}
```

Finally, here is a `main` method that creates and displays a `ButtonFrame`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonTest

{  public static void main(String[] args)
   {  JFrame frame = new ButtonFrame();
      frame.show();
   }
}
```

8.2.2 Three buttons

Suppose we want to have a panel with three buttons, labelled Red, Blue and Yellow, so that clicking each button results in the background colour changing to the appropriate colour. We can modify the earlier ButtonPanel to contain three buttons.

The ActionListener associated with each button has to be able to inform the ButtonPanel to change its background colour. The simplest solution is to make the ButtonPanel itself the ActionListener for all three buttons. However, we now have a single ActionListener handling events from multiple sources. In ActionPerformed, we can use the getSource() method to find out the source of the event and then take appropriate action.

Here is the revised code for ButtonPanel for this example. The code for ButtonFrame does not change.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonPanel extends JPanel implements ActionListener{

    private JButton yellowButton;    // Panel has three buttons
    private JButton blueButton;
    private JButton redButton;

    public ButtonPanel(){
        yellowButton = new JButton("Yellow");
        blueButton = new JButton("Blue");
        redButton = new JButton("Red");

        yellowButton.addActionListener(this);  // ButtonPanel is the
        blueButton.addActionListener(this);    //   listener for all
        redButton.addActionListener(this);     //   three buttons

        add(yellowButton);
        add(blueButton);
        add(redButton);
    }

    public void actionPerformed(ActionEvent evt){
        Object source = evt.getSource();       // Find the source of the
                                               //   event
        Color color = getBackground();         // Get current background
                                               //   colour
        if (source == yellowButton) color = Color.yellow;
        else if (source == blueButton) color = Color.blue;
        else if (source == redButton) color = Color.red;

        setBackground(color);
        repaint();
    }
}
```

Some more examples can be found in Appendix A.

8.3 The event queue

Recall that the operating system interacts with the run-time support for the programming language to pass on events such as keystrokes and mouse movements. These are automatically resolved into high level events in terms of the components being manipulated by the program, such as buttons, etc.

Events are despatched sequentially by the run-time support system to the program. Thus, if the user clicks a button such as `redraw` in a graphics application, until the redrawing is completed the application cannot respond to other events. However, the user is clearly free to move the mouse and press some keys while the redrawing is in progress. These events are stored in an *event queue* and processed in turn.

One advantage of storing events in a queue is that they can be optimized. For instance, if there are three mouse movements while the redrawing is in progress, it is sufficient to record the position of the mouse at the end of the last of the three movements in the queue.

What events go in the queue? We have so far indicated that the program receives only high level, or "semantic", events such as "button `b` was pushed". These are synthesized by the run-time support system from the low level events that it receives from the operating system.

However, a program may also like to receive low level events such as mouse movements and key strokes. Consider a drawing application where the user can draw a line segment by using the mouse to click both end points. After selecting the first point, it would be complicated if the user could wander off and do other things before identifying the second point. In such a situation, after being notified that the first point has been selected, which is perhaps a high level event, the program can go into a loop where it explicitly "consumes" events in the queue until it sees the next mouse click within the drawing area. All irrelevant intervening events are discarded. Thus, by examining low level events, the program can "grab" control of the user and force the user to act in a predictable way.

The low level support for event management in Java is done by a portion of the AWT system. The AWT system does in fact insert both low level and high level events in the queue. The low level events are essentially those received from the operating system, but these may be combined and optimized as indicated earlier. In addition, AWT keeps track of how high level components are affected by these low level events and inserts appropriate high level events into the queue.

Low level events have listener interfaces, just like high level ones. Thus, normally, a Java program interacts with the event queue implicitly, by defining appropriate listeners for the various events of interest. When an event reaches the head of the queue, it activates all the listeners that exist for it. If there are no listeners active, the event is discarded.

In addition, a Java program can also explicitly interact with the event queue. Java has a built-in class called `EventQueue`. An object of this class can hold a pointer to the current system event queue. This is done as follows.

```
EventQueue evtq = Toolkit.getDefaultToolkit().getSystemEventQueue();
```

It is then possible to consume an event from the head of the event queue explicitly, as follows.

```
AWTEvent evt = eq.getNextEvent();
```

(It is also possible to "peek" at the event at the head of the queue without consuming it.)

We can use this explicit event queue manipulating facility to write event grabbing code such as the line drawing application discussed earlier that insists on a second point being selected after the first point has been placed.

Dually, we can also add events to the queue from within the program by writing something like

```
evtq.postEvent(new ActionEvent(.....));
```

Thus, when one button is pressed, one of the effects could be to internally generate an event corresponding to another button being pressed. For instance, a button labelled `Exit` might generate a frame-closing event for its enclosing frame which, when handled, will shut down the application.

8.4 Custom events

Just like we can add new types of exceptions in a Java program, we can also define new types of events. Let us look at how to add a new type of event called `TimerEvent` to be generated by a `Timer` object.

Events in Java (generally) extend the class `AWTEvent`. Each distinct type of event has a unique numeric identifier. We can thus write

```
class TimerEvent extends AWTEvent{

   public static final int
```

```
        TIMER_EVENT = AWTEvent.RESERVED_ID_MAX + 5555;

    public TimerEvent(Timer t) {
      super(t, TIMER_EVENT);
    }
 }
```

Here, we use a language defined constant that delimits the numeric identifiers in use for built-in events to ensure that our new event does not clash with any of the built-in events. The constructor essentially sets two quantities: the source of the event and the (numeric) type of the event.

Next, we define an interface that specifies the functions that a listener for `TimerEvent` must implement. This might be as follows

```
interface TimerListener extends EventListener{
   public void timeElapsed(TimerEvent evt);
}
```

Now, we can make the class `Timer` generate such events by explicitly manipulating the event queue, as described earlier.

```
class Timer extends Component implements Runnable{
   // Runnable is to ensure that Timer can run concurrently
   ...
   private static EventQueue evtq =
          Toolkit.getDefaultToolkit().getSystemEventQueue();
   // There is only one copy of the EventQueue, shared by all Timers

   public Timer(){
     ...
   }

   public void f(){
      ...
      // Generate a TimerEvent when done
      TimerEvent te = new TimerEvent(this);
      evtq.postEvent(te);
   }
}
```

Of course, we need a class that listens to TimerEvents, so we have

```
class MyClass implements TimerListener{
   ...
```

```
     public void timeElapsed(TimerEvent evt){
       ...
     }
   }
```

We need to inform the Timer object of the existence of a specific listener object from MyClass. For this, we have to add the following in Timer.

```
   class Timer extends Component implements Runnable{
     ...
     private TimerListener listener;

     public addListener(TimerListener t){
       listener = t;
     }
     ...
   }
```

In other words, it is the duty of the Timer object to keep track of the listeners it has to notify. This is the general Java philosophy, which is why we have methods like addActionListener(..) for JButtons etc.

Assume now that we have code like the following:

```
    ...
    Timer t = new Timer();
    MyClass m = new MyClass(..);
    t.addListener(m);
    t.f();
    ...
```

At the end of t.f(), how does t know what it has to do to notify m? The answer is that it does not. This has to be programmed in the definition of Timer. What the AWT system provides is the name of a fixed function called processEvent that will be called within an object whenever it generates an AWTEvent. We can now add the following code in Timer.

```
   class Timer extends Component implements Runnable{
     ...

     public void processEvent(AWTEvent evt)
     {  if (evt instanceof TimerEvent)
        {  if (listener != null)
              listener.timeElapsed((TimerEvent)evt);
        }
        else super.processEvent(evt);
```

```
    }
    ...
}
```

Now, all the pieces are in place. When a `Timer t` generates a `TimerEvent`, AWT calls the (fixed) method `processEvent` within `t`. This provides the explicit code with which to call any listener associated with `t`. Notice that instead of one `TimerListener`, we could have defined `Timer` to store an array of listeners and notified each of them in turn, thus supporting multicasting.

The same general architecture drives the functioning of builtin objects like `JButton` etc. Externally, it appears as though the run-time environment "knows" that when a `JButton` is pressed, the correct method to invoke in an `ActionListener` is `actionPerformed`. However, all that the button press event does is to invoke `processEvent` in the `JButton` object that is pressed. The definition of `JButton`, like the definition that we have just given for `Timer`, includes all the necessary information about its listeners and the methods to use to notify these listeners that the button has been pressed.

Part IV

Reflection

Chapter 9

Reflection

A programming language is said to be *reflective*[1] if it can represent (part of its) state as data and manipulate it. There are two components involved in reflection:

- **Introspection**: The ability of a program to observe and therefore reason about its own state.
- **Intercession**: The ability of a program to modify its execution state or alter its own interpretation or meaning.

Constructions such as the following

```
Employee e = new Manager(...);
...
if (e instanceof Manager){
   ...
}
```

are very simple examples of reflection at work. We can, at run time, check the actual type of an object.

However, notice that in this case we know the type that we want to check in advance—both `Employee` and `Manager` are already defined when this code is written. This is important because the second argument to `instanceof` must be a fixed name for a class, not a "class variable".

What if we wanted to check whether two pointers are the same at run-time? We would like to write a function like:

```
public static boolean classequal(Object o1, Object o2){
  ...
  // return true iff o1 and o2 point to objects of same type
  ...
}
```

[1] The word *reflection* is used in the philosophical sense, meaning "to think seriously".

To implement this using instanceof, we would have to check this for every possible class that we know whether o1 and o2 are both instances of the same class. This is infeasible for two reasons:

- We could not reasonably expect to enumerate all the classes defined in the system, even if this set was fixed.
- More critically, we would like to use this function in any Java environment, including those in which classes that are yet to be defined may be loaded!

It turns out that Java allows us to extract the class of an object at runtime, using the method getClass(), defined in the Java reflection package java.lang.reflect. Thus, we write the following code to implement the classequal we were looking at earlier.

```
import java.lang.reflect.*;

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        return (o1.getClass() == o2.getClass());
    }
}
```

What values are being compared in the test o1.getClass() == o2.getClass()? In other words, what does getClass() return? For the purpose of classequal, it would be sufficient to have a naïve return value, such as a String containing the name of the the class. In fact, getClass() does much more. It returns a representation of the class of the object on which it is invoked. This means that there is a built in class in Java that can store such a representation. This built in type is called, somewhat confusingly, Class! To make this explicit, we can rewrite classequal as follows.

```
import java.lang.reflect.*;

class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        Class c1, c2;
        c1 = o1.getClass();
        c2 = o2.getClass();
        return (c1 == c2);
    }
}
```

Notice that we are representing dynamic information about the state of the system (the runtime class of `o1` and `o2`) in terms of data `c1` and `c2` that can be manipulated by the program. This process of encoding execution state as data is called *reification*.

Having got a representation for a class in a `Class` object, we can do various things with it. For instance, we can create new instances of this class.

```
...
Class c = obj.getClass();
Object o = c.newInstance();  // Create a new object of same type as obj
...
```

Another way of obtaining a class object is to supply the name of the class as a string to the static function `forName` defined in `Class`.

```
...
String s = "Manager".
Class c = Class.forName(s);
Object o = c.newInstance();
...
```

We could, of course, simplify this into a single statement as follows:

```
...
Object o = Class.forName("Manager").newInstance();
...
```

Once we have a class stored in an object of type `Class`, we can find out details about its constructors, methods and fields (instance variables). A constructor, method or field is itself a complex quantity. For instance, to full describe a constructor we need to provide the types of arguments. For a method we need the arguments and the return type. For all three, we need the modifiers (`static`, `private` etc). Thus, it makes sense to have additional classes called `Constructor`, `Method` and `Field` to store information about an individual constructor, method or field. Given an object `c` of type `Class`, we can invoke the functions `getConstructors()`, `getMethods()` and `getFields()` to obtain the list of constructors, methods and fields of `c`. Each of these functions returns an array. So, for instance, we could write

```
...
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
Method[] methods = c.getMethods();
Field[] fields = c.getFields();
...
```

Not surprisingly, we can now operate on each of `Constructor`, `Method` and `Field` to get further details of these objects. For instance, the list of arguments to a constructor or a method is a list of types, or, in other words, an array of `Class[]`. Thus, we can write

```
...
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
for (int i = 0; i < constructors.length; i++){
  Class params[] = constructors[i].getParameterTypes();
  ..
}
```

We can also invoke methods and examine/set values of fields.

```
...
Class c = obj.getClass();
..
Method[] methods = c.getMethods();
Object[] args = { ... }       // construct an array  of arguments
methods[3].invoke(obj,args);  // invoke method stored in methods[3]
                              //  on obj with arguments args
...
Field[] fields = c.getFields();
Object o =  fields[2].get(obj);  // get the value of fields[2]
                                 // from obj
...
fields[3].set(obj,value);        // set the value of fields[3]
                                 // in obj to value
...
```

Should it be possible to find out about private constructors, methods and fields of a class? The functions `getConstructors()`, ...only return publicly defined values. However, Java does permit us to access private components using variants of these functions called `getDeclaredConstructors()`; `getDeclaredMethods()` and `getDeclaredFields()`. Of course, there is an issue of security involved here. In general, Java allows us to find out about private methods or fields in a class, but the default security level will not allow us to invoke private methods or access/modify private fields.

The reflective capabilities of Java allow us to write versatile code, such as the Java programming environment *BlueJ*[2] that provides an interactive interface to define Java classes, create objects, invoke methods on these objects, examine the state of these objects etc. The important fact is that *BlueJ* does not have to implement a nested virtual machine to act as

[2] Available at `http://www.bluej.org`

a Java “interpreter”. Using the reflective capabilities of Java, *BlueJ* can directly perform its manipulations using the JVM within which it is itself running!

Of course, one might ask for even more—there is no way to directly create or modify new classes by writing something like

```
Class c = new Class(....);
```

Creating class objects is done indirectly by the class loader, which is outside the scope of this discussion. Nevertheless, Java does provide a rather powerful set of tools for reflective programming.

Part V

Concurrent programming

Chapter 10

Concurrent Programming

In a distributed computing system, it is natural to regard the nodes as independent processes that execute concurrently. In such a system, agents communicate with each other to coordinate their activities. Communication may be through synchronization or message-passing. Synchronization implies a simultaneous two-way exchange of information (like a telephone call) while message-passing involves one-way transfers that require acknowledgments for the sender to be sure that a message has reached the recipient (like a normal letter sent by post).

However, even in a single processor system, it is often convenient to regard different components of a program as executing in parallel. Consider, for instance, an interactive application like a web browser. When you are downloading a page that takes a long time to retrieve, you can usually press the `Stop` button to terminate the download. When programming a browser with such a capability, the most natural model is to regard the download component and the user-interface component (that reacts to button clicks) as separate processes. When the user-interface process detects that the `Stop` button has been clicked, it notifies the download process that it can terminate its activity.

On a single processor, the run-time environment will allocate time-slices to each of these "logical" processes. These time-slices will be scheduled so that all the concurrent processes get a chance to execute. However, there is no guarantee about the relative speeds at which these processes will run—one process may get 10 more time-slices than another in a given interval.

Normally, each concurrent process comes with its own local variables. Thus, when the run-time environment switches from one process to another, it has to save the state of the first process and load the suspended state of the second one. Often, however, it is simpler to assume that all the concurrent processes share the same set of variables. Thus, the processes interact via a global "shared memory". This makes it possible to switch from one process to another relatively easily, without an elaborate context switch involving all the variables defined in the processes. In the literature, these kinds of processes that share a global memory are often called "threads" (short for "threads of execution").

We shall study concurrent programming in the framework of threads that operate with a global shared memory. In the rest of these notes, the words thread and process will be used synonymously.

10.1 Race conditions

Shared variables provide a simple mechanism for communication between processes. For instance, in the example of the browser discussed earlier, we could have a boolean variable that indicates whether the download should be interrupted. This variable is initially false. When the user-interface thread detects that the ”Stop” button has been clicked, it sets the variable to true. Periodically, the download thread examines the value of this variable. If it finds that the variable has been set to true, it aborts the current transfer.

Once we have shared variables, it becomes important to incorporate a mechanism for consistent update of such variables by concurrent threads. Consider a system in which we have two threads that update a shared variable `n`, as follows:

```
Thread 1                          Thread 2

...                               ...
m = n;                            k = n;
m++;                              k++;
n = m;                            n = k;
...                               ...
```

Under normal circumstances, after these two segments have executed, we would expect the value of `n` to have been incremented twice, once by each thread. However, because of time-slicing, the order of execution of the statements may be as follows;

```
Thread 1: m = n;
Thread 1: m++;
Thread 2: k = n;   // k gets the original value of n
Thread 2: k++;
Thread 1: n = m;
Thread 2: n = k;   // Same value as that set by Thread 1
```

In this sequence, the increments performed by the two threads overlap and the final value of `n` is only one more than its initial value. This kind of inconsistent update, whose effect depends on the exact order in which concurrent threads execute, is known as a *race condition*.

Here is a more complex example of the same phenomenon. Suppose we have a shared array

```
double accounts[100]
```

that holds the current balance for 100 bank accounts. Suppose we have have two functions as follows:

```
// transfer "amount" from accounts[source] to accounts[target]
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    return false;
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}

// compute the total balance across all accounts
double audit(){
  double balance = 0.00;
  for (int i = 0; i < 100; i++){
    balance += accounts[i];
  }
  return balance;
}
```

Now, suppose we execute these functions in concurrent threads, as follows:

```
Thread 1                              Thread 2
...                                   ...
status = transfer(500.00,7,8);        print (audit());
...                                   ...
```

Suppose that Thread 2 gets access to `accounts[7]` and `accounts[8]` after Thread 1 has debited `accounts[7]` but before it has credited `accounts[8]`. The audit will then report that a sum of 500.00 has been "lost" from the accounts in the bank, because it uses the updated value of `accounts[7]` but the old value of `accounts[8]`.

The situation is even more complicated than this. Even if Thread 1 executes both the debit and the credit without interruption, it is possible that Thread 2 reads `accounts[7]` before the transfer and `accounts[8]` after the transfer, thereby recording an excess of 500.00 in the total balance.

For these functions to execute concurrently in parallel, both `transfer(...)` and `audit()` should execute from beginning to end without interruption. Stated another way, it should never be the case that the current control point in one thread is within `transfer(...)` while in the other thread it is within `audit()`.

All of these examples can be formulated as instances of the familiar problem of *mutual exclusion* to *critical regions* of program code, which is well-studied, for instance, in the design of operating systems.

10.2 Protocols based on shared variables

One way to solve the mutual exclusion problem is to develop a protocol using auxiliary shared variables. For instance, we could use a variable `turn` that takes values 1 and 2 to indicate whose turn it is to access the critical region, as follows.

```
Thread 1                          Thread 2


...                               ...
while (turn != 1){                while (turn != 2){
  // "Busy" wait                    // "Busy" wait
}                                 }
// Enter critical section         // Enter critical section
   ...                               ...
// Leave critical section         // Leave critical section
turn = 2;                         turn = 1;
...                               ...
```

Let us assume that turn is initialized to either 1 or 2 (arbitrarily) and there is no other statement that updates the value `turn` other than the two assignments shown above. It is then clear that both Thread 1 and Thread 2 cannot simultaneously enter their critical sections—if Thread 1 has entered its critical section, the value of `turn` is 1 and hence Thread 2 is blocked until Thread 1 exits and sets `turn` to 2. A symmetric argument holds if Thread 1 tries to enter the critical section while Thread 2 is already in its critical section. If both threads simultaneously try to access the critical section, exactly one will succeed, based on the current value of `turn`.

Notice that this solution does not depend on any atomicity assumptions regarding assigning a value to turn or testing the current value of turn.

However, there is one serious flaw with this solution. When Thread 1 executes its critical section it sets `turn` to 2. The only way for Thread 1 to be able to reenter the critical section is for Thread 2 to reset `turn` to 1. Thus, if only one thread is active, it cannot enter the critical section more than once. In this case, the thread is said to *starve*.

Another solution is to maintain two boolean variables, `request_1` and `request_2`, indicating that the corresponding thread wants to enter its critical section.

```
Thread 1                          Thread 2


...                               ...
request_1 = true;                 request_2 = true;
while (request_2){                while (request_1)
  // "Busy" wait                    // "Busy" wait
}                                 }
// Enter critical section         // Enter critical section
```

```
   ...                              ...
// Leave critical section        // Leave critical section
request_1 = false;               request_2 = false;
...                              ...
```

Here we assume that `request_1` and `request_2` are initialized to `false` and, as before, these variables are not modified in any other portion of the code. Once again, it is easy to argue that both threads cannot simultaneously be in their critical sections—when Thread 1 is executing its critical section, for instance, `request_1` is true and hence Thread 2 is blocked. Also, if only one thread is alive, it is still possible for that thread to repeatedly enter and leave its critical section since the request flag for the other thread will be permanently false. However, if both threads simultaneously try to access the critical region, they will both set their request flags to true and there will be a deadlock where each thread is waiting for the other thread's flag to be reset to false.

Peterson [1981] found a clever way to combine these two ideas into a solution that is free from starvation and deadlock. Peterson's solution involves both the integer variable `turn` (which takes values 1 or 2) and the boolean variables `request_1` and `request_2`.

```
Thread 1                         Thread 2

...                              ...
request_1 = true;                request_2 = true;
turn = 2;                        turn = 1;
while (request_2 &&              while (request_1 &&
       turn != 1){                      turn != 2){
  // "Busy" wait                   // "Busy" wait
}                                }
// Enter critical section        // Enter critical section
   ...                              ...
// Leave critical section        // Leave critical section
request_1 = false;               request_2 = false;
...                              ...
```

If both threads try to access the critical section simultaneously, the variable `turn` determines which process goes through. If only one thread is alive, the request variable of the other thread is stuck at false and the value of `turn` is irrelevant. To check that mutual exclusion is guaranteed, suppose Thread 1 is already in its critical section. If Thread 2 then tries to enter, it first sets `turn` to 1. While Thread 1 is active, `request_1` is true. Thus, Thread 2 has to wait until Thread 1 finishes and turns off `request_1`. A symmetric argument holds when Thread 2 is in the critical region and Thread 1 tries to enter.

10.3 Programming primitives for mutual exclusion

Peterson's algorithm does not generalize easily to a situation where there are more than two concurrent processes. For n processes, there are other solutions, such as Lamport's Bakery algorithm. However, the problem with these protocols is that, in general, they have to be designed anew for different situations and checked for correctness.

A better solution is to include support in the programming language for mutual exclusion. The first person to propose such primitive was Edsger Dijkstra, who suggested a new datatype called a semaphore.

A *semaphore* is an integer variable that supports an atomic test-and-set operation. More specifically, if a `S` is a variable of type semaphore, then two atomic operations are supported on `S`: `P(S)` and `V(S)`. (The letters `P` and `V` come from the Dutch words *passeren*, to pass, and *vrygeven*, to release.)

The operation `P(S)` achieves the following in an atomic manner:

```
if (S > 0)
  decrement S;
else
  wait for S to become positive;
```

The operation `V(S)` is defined as follows:

```
if (there are threads waiting for S to become positive)
  wake one of them up; //choice is nondeterministic
else
  increment S;
```

Using semaphores, we can now easily program mutual exclusion to critical sections, as follows:

```
Thread 1                           Thread 2

...                                ...
P(S);                              P(S);
// Enter critical section          // Enter critical section
   ...                                ...
// Leave critical section          // Leave critical section
V(S);                              V(S);
...                                ...
```

Mutual exclusion, starvation freedom and deadlock freedom are all guaranteed by the definition of a semaphore.

One problem with semaphores are that they are rather "low-level". The definition of a semaphore is orthogonal to the identification of the critical region. Thus, we connect critical regions across threads in a somewhat arbitrary fashion by using a shared semaphore. Correctness requires cooperation from all participating threads in resetting the semaphore. For instance, if any one thread forgets to execute `V(S)`, all other threads get blocked. Further, it is not required that each `V(S)` match a preceding `P(S)`. This could lead to a semaphore being "preloaded" to an unsafe value.

10.4 Monitors

Since critical regions arise from the need to permit consistent updates to shared data, it seems logical that information about conflicting operations on data should be defined along with the data, using a class-like approach. This is the philosophy underlying *monitors*, a higher-level approach to providing programming support for mutual exclusion. Monitors were proposed by Per Brinch Hansen and Tony Hoare.

Roughly speaking, a monitor is like a class description in an object-oriented language. At its heart, a monitor consists of a data definition, to which access should be restricted, along with a list of functions to update this data. The functions in this list are assumed to all be mutually exclusive—that is, at most one of them can be active at any time. The monitor guarantees this mutual exclusion by making other function calls wait if one of the functions is active.

For instance, here is a monitor that handles the bank account example from the previous lecture:

```
monitor bank_account{

  double accounts[100];

  // transfer "amount" from accounts[source] to accounts[target]
  boolean transfer (double amount, int source, int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  // compute the total balance across all accounts
  double audit(){
    double balance = 0.0;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
```

```
        return balance;
      }
  }
```

By definition, the functions `transfer(...)` and `audit()` are assumed to require mutually exclusive access to the array `accounts[]`. Thus, if one process is engaged in a transfer and a second process wants to perform an audit, the second process is suspended until the first process finishes.

Thus, there is an implicit "external" queue associated with the monitor, where processes that are waiting for access are held. We use the word queue but, in practice, this is just a set. No assumption is made about the order in which waiting processes will get access to the monitor.

The capabilities we have described so far for monitors are not as general as, say, semaphores. Consider for instance the following scenario.

We have three bank accounts, `i`, `j` and `k` and we want to perform the following two operations concurrently.

```
transfer(500.00,i,j);
transfer(400.00,j,k);
```

If the balance in `i` is greater than 500.00, this pair of transfers should always succeed. However, if `accounts[j]` is not at least 400.00 initially, there is a possibility that the second transfer will be scheduled before the first and will fail.

We could try to get around this by reprogramming the transfer function as follows:

```
// transfer "amount" from accounts[source] to accounts[target]
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    // wait for another transaction to transfer money
    // into accounts[source]
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

The problem is that all other processes are blocked out by the monitor while this process waits. We thus need to augment the monitor definition with the ability to suspend a process and relinquish the monitor.

A process that suspends itself is waiting for the data in the monitor to change its state. It does not make sense to place such a process in the same queue as one that is waiting (unconditionally) to enter the monitor for the first time. Instead, it makes sense to a second queue with a monitor where suspended processes wait, which we shall call an "internal" queue, to distinguish it from the "external" queue where blocked processes wait.

We then need a dual operation to wake up a suspended process when the state changes in a way that may be of benefit to the suspended process—in this example, when another transfer succeeds.

Using these ideas, we can now rewrite the transfer function as follows, where `wait()` suspends a process and `notify()` wakes up a suspended process.

```
// transfer "amount" from accounts[source] to accounts[target]
boolean transfer (double amount, int source, int target){
  if (accounts[source] < amount){
    wait();
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

We now have to deal with another tricky problem: what happens when a `notify()` is issued? The notifying process is itself still in the monitor. Thus, the process that is woken up cannot directly execute.

At least three different types of notification mechanisms have been identified in the literature.

The simplest is one called *signal and exit.* In such a setup, a notifying process must immediately exit the monitor after notification and hand over control to the process that has been woken up. This means that `notify()` must be the last instruction that it executes. We shall see later that we might use multiple internal queues in a monitor, each of which is notified independently. With signal and exit, it is not possible to notify multiple queues because the first call to `notify()` must exit the monitor.

Another mechanism is called *signal and wait.* In this setup, the notifying process and the waiting process swap roles, so the notifying process hands over control of the monitor and suspends itself. This is not very commonly used.

The last mechanism is *signal and continue.* Here, the notifying process continues in the monitor until it completes its normal execution. The woken up process(es) shift from the internal to the external queues and are in contention to be scheduled when access to the monitor is relinquished by the notifying process. However, no guarantee is given that one of the newly awakened processes will be the next to enter the monitor—they may all be blocked out by another process that was already waiting in the external queue.

When a suspended process resumes execution, there is no guarantee that the condition it has been waiting for has been achieved. For instance, in the example above, `transfer` is waiting for an amount of at least 400 to be transferred into `accounts[j]`. Any subsequent successful transfer will potentially `notify()` this waiting process, including transfers to accounts other than `j`. Even if the transfer is into `accounts[j]`, the amount might be less than the 400 threshold required by the waiting copy.

The moral of the story is that a waiting condition should check the state of the data before proceeding because the data may still be in an undesirable state when it wakes up. We should rewrite the earlier code as:

```
// transfer "amount" from accounts[source] to accounts[target]
boolean transfer (double amount, int source, int target){
  while (accounts[source] < amount){
    wait();
  }
  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

In other words, the `if` is replaced by a `while` so that the first thing done after waking up from `wait()` is to check the condition again, before proceeding.

In this example, clearly not every successful transfer is relevant to a waiting process. The only relevant transfers are those that feed into the account whose balance is low. Thus, instead of a single internal queue, it makes sense to have multiple queues—in this case, one queue for each account (in other words, an array of queues). Then, a successful transfer will notify only the queue pertaining to the relevant account.

This means that we have to extend the monitor definition to include a declaration of the internal queues (sometimes called *condition variables*) associated with the monitor. In the example below, we assume we can declare an array of queues. In practice, this may not be possible—we may have to explicitly assign a separate name for each queue.

```
monitor bank_account{

  double accounts[100];

  queue q[100];  // one internal queue for each account

  // transfer "amount" from accounts[source] to accounts[target]
  boolean transfer (double amount, int source, int target){
    while (accounts[source] < amount){
      q[source].wait();  // wait in the queue associated with source
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    q[target].notify();  // notify the queue associated with target
    return true;
  }
```

```
      // compute the total balance across all accounts
      double audit(){ ...}

  }
```

If you notify() a queue that is empty, it has no effect. This is more robust than the V(S) operation of a semaphore that increments S when nobody is waiting on S.

10.5 Monitors in Java

Java implements monitors, but with a single queue. The queue is "anonymous" (but is really implicitly named by the object in which it is defined, as we shall see).

Any class definition can be augmented to include monitor like behaviour. We add a qualifier synchronized to any method that requires exclusive access to the instance variables of the class. Thus, if we have two or more functions that are declared to be synchronized and one of them is currently being executed, any attempt to enter another synchronized method will be blocked and force the thread into the external queue. In Java terminology, there is a lock associated with the object. Only one thread can own the lock at one time and only the thread that owns the lock can execute a synchronized method. At the end of the synchronized method, the thread relinquishes the lock.

Inside a synchronized method, the wait() statement behaves like the normal wait() statement in a monitor, except that there is only one (unnamed) queue that the process can wait on. The corresponding wakeup call should be called notify() but in Java, for some obscure reason, there are two functions notify() and notifyAll(). The function notify() signals only one of the waiting processes, at random. Instead, notifyAll() signals all waiting processes (as one would expect) and this should be used by default.

Java monitors use the *signal and continue* mechanism, so the notifying process can continue execution after sending a wakeup signal and the processes that are woken up move to the external queue and are in contention to grab the object lock when it next becomes available.

The Java implementation of the bank account class looks pretty much like the monitor description given earlier. One difference is that it can include non-synchronized methods, which do not require having control of the object lock.

```
    public class bank_account{

      double accounts[100];

      // transfer "amount" from accounts[source] to accounts[target]
      public synchronized boolean
          transfer (double amount, int source, int target){
```

```
        while (accounts[source] < amount){
          wait();
        }
        accounts[source] -= amount;
        accounts[target] += amount;
        notifyAll();
        return true;
      }

      // compute the total balance across all accounts
      public synchronized double audit(){
        double balance = 0.0;
        for (int i = 0; i < 100; i++){
          balance += accounts[i];
        }
        return balance;
      }

      // a non-synchronized method
      public double current_balance(int i){
        return accounts[i];
      }
  }
```

In addition to synchronized methods, Java has a slightly more flexible mechanism. Every object has a lock, so an arbitrary block of code can be synchronized with respect to any object, as follows:

```
  public class XYZ{

  Object o = new Object();

  public int f(){
    ..
    synchronized(o){
       ...
    }
  }

  public double g(){
    ..
    synchronized(o){
       ...
    }
  }
```

Now, `f()` and `g()` can both begin to execute in parallel in different threads, but only one thread at a time can grab the lock associated with `o` and enter the block enclosed by `synchronized(o)`. The other will be placed in the equivalent of an "external queue" for the object `o`, while waiting for the object lock.

In addition, there is a separate "internal queue" associated with `o`, so one can write.

```
  public class XYZ{

  Object o = new Object();

  public int f(){
    ..
    synchronized(o){
       ...
       o.wait();   // Wait in the internal queue attached to "o"
       ...
    }
  }

  public double g(){
    ..
    synchronized(o){
       ...
       o.notifyAll();   // Wake up the internal queue attached to "o"
       ...
    }
  }
```

Notice that we can "rewrite" completely a synchronized method of the form

```
  public synchronized double h(){
    ...
  }
```

as an externally unsynchronized method that is internally synchronized on `this` as follows:

```
  public double h(){
    synchronized(this){
      ...
    }
  }
```

Also, the "anonymous" calls `wait()` and `notify()/notifyAll()` are actually the normal object-oriented abbreviations for `this.wait()` and `this.notify()/this.notifyAll()`.

Actually, `wait()` throws an exception `InterruptedException` (that we shall examine when we look at how to define threads in Java), so more correctly we should encapsulate calls to `wait()` (or `o.wait()`) as follows:

```
try{
  wait();
}
catch (InterruptedException e);
```

Also, it is a mistake to use `wait()` or `notify()/notifyAll()` other than in a synchronized method. This will throw an `IllegalMonitorStateException`. Similarly, it is an error to to use `o.wait()`, `o.notify()` or `o.notifyAll()` other than within a block synchronized on `o`.

10.6 Java threads

The simplest way to define concurrent threads in Java is to have a class extend the built-in class `Thread`. Suppose we have such a class, as follows, in which we define a function `run()` as shown:

```
public class Parallel extends Thread{

  private int id;

  public Parallel(int i){  // Constructor
    id = i;
  }

  public void run(){
    for (int j = 0; j < 100; j++){
      System.out.println("My id is "+id);
      try{
        sleep(1000);        // Go to sleep for 1000 ms
      }
      catch(InterruptedException e){}
    }
  }
}
```

Then, we can created objects of type `Parallel` and "start" them off in parallel, as follows:

```
public class TestParallel {

  public static void main(String[] args){

    Parallel p[] = new Parallel[5];

    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       p[i].start();       // Start off p[i].run() in concurrent thread
    }

}
```

The call `p[i].start()` initiates the function `p[i].run()` in a separate thread. (Note that if we write `p[i].run()` instead of `p[i].start()` we just execute `p[i].run()` like any other function in the current thread—that is, control is transferred to `p[i].run()` and when it finishes, control passes back to the caller.)

Notice the function `sleep(..)` used in `run()`. This is a static function defined in `Thread` that puts the current thread to sleep for the number of milliseconds specified in its argument. Any thread can put itself to sleep, not just one that extends `Thread`. In general, one would have to write `Thread.sleep(..)` if the current class does not extend `Thread`. Observe that `sleep(..)` throws `InterruptedException` (like `wait()`, discussed earlier).

Of course, given the single inheritance mechanism of Java, we cannot always extend `Thread` directly. An alternative is to implement the interface `Runnable`. A class that implements `Runnable` is defined in the same way as one that extends `Thread`—we have to define a function `public void run()`... However, to invoke this in parallel, we have to explicitly create a `Thread` from the `Runnable` object by passing the reference to the object to the `Thread` constructor. Here is how we would rework the earlier example.

First the `Runnable` class:

```
public class Parallel implements Runnable{  // only this line
                                            //   has changed
  private int id;
  public Parallel(int i){ ... } // Constructor
  public void run(){ ... }

}
```

Then, the class that uses the `Runnable` class.

```
public class TestParallel {

  public static void main(String[] args){
    Parallel p[] = new Parallel[5];
    Thread t[]   = new Thread[5];

    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       t[i] = new Thread(p[i]);  // Make a thread t[i] from p[i]
       t[i].start();       // Start off p[i].run() in concurrent thread
                           // Note: t[i].start(), not p[i].start()
    }

}
```

A thread can be in one of four states:

- **New**: When it has been created but not `start()`ed.
- **Runnable**: When it has been `start()`ed and is available to be scheduled.

  A Runnable thread need not be "running"—it may be waiting to be scheduled. No guarantee is made about how scheduling is done but almost all Java implementations now use time-slicing across running threads.
- **Blocked**: The thread is not available to run at the moment. This could happen for three reasons:

  1. The thread is in the middle of a `sleep(..)`. It will then get unblocked when the sleep timer expires.
  2. The thread has suspended itself using a `wait()`. It will get unblocked by a `notify()` or `notfifyAll()`.
  3. The thread is blocked on some input/output. It gets unblocked when the i/o succeeds.
- **Dead**: This state is reached when the thread terminates.

10.7 Interrupts

A thread can be interrupted by another thread using the function `interrupt()`. Thus, in our earlier example, we can write

```
p[i].interrupt();
```

to interrupt thread p[i]. This raises an InterruptedException, which, as we saw, we must catch if the thread is in sleep() or wait().

However, if the thread is actually running when the interrupt arrives, no exception is raised! To identify that an interrupt arrived in such a situation, the thread can check its interrupt status using the function interrupted(), which returns a boolean and clears the interrupt status back to false. Thus, if we want to trap interrupts in p[i].run() both at the sleep(..) and elsewhere, we should rewrite run() as follows:

```
     public void run(){
       try{
         j = 0;
         while(!interrupted() && j < 100){
           System.out.println("My id is "+id);
           sleep(1000);        // Go to sleep for 1000 ms
           j++;
       }
       catch(InterruptedException e){}
       }
     }
   }
```

It is also possible to check another thread's interrupt status using isInterrupted(). Thus t.isInterrupted() returns a boolean. However, it does not clear the interrupt flag, unlike interrupted(). Actually, interrupted() is a static function that makes use of isInterrupted() to check the flag of the current thread and then clears its interrupt flag.

Another useful function to spy on the state of another thread is isAlive(). t.isAlive() returns true if the thread t is Runnable or Blocked and false if t is New or Dead. Notice that you cannot use isAlive() to distinguish whether a thread is Runnable or Blocked.

For historical reasons, Java also supports a method stop() to kill another thread and suspend() and resume() to unilaterally suspend and resume a thread (which is not the same as the wait() and notify()/notifyAll() construction). However, these are not supposed to be used because they frequently lead to unpredictable conditions (when a thread dies abruptly) or deadlocks (when suspended threads are not properly resumed).

Part VI

Functional programming

Chapter 11

Functional programming in Haskell

Haskell is a functional programming language with a strong notion of type (like ML, and unlike the family of languages based on Lisp). Haskell differs from ML in the strategy it uses for evaluating functional expressions—as we shall see, Haskell uses an "outside-in", "lazy" approach while ML uses an "inside-out", "eager" approach.

11.1 Types

Haskell has basic built-in scalar types `Int`, `Float`, `Bool` and `Char`. In addition, given a type `T`, you can form a list of elements of type `T`. The list has type `[T]`. Also, given types `T1`, `T2`, ..., `Tk`, you can form k-tuples of type `(T1,T2,...,Tk)`.

Functions have types—a function whose input type is `T1` and output type is `T2` has type `T1 -> T2`. For instance, the factorial function has type

```
factorial :: Int -> Int
```

Note the syntax `::` to denote the type of a function.

What about a function such as `max`, that computes the maximum of two integer inputs? A natural definition of its type would be

```
max :: (Int x Int) -> Int
```

where `(Int x Int)` denotes the set of all pairs of type `Int`. In general a function of n arguments would have type `(T1 x T2 x ...  x Tn) -> T` where `Ti` is the type of argument i and `T` is the return type. However, for this we have to record with each function its arity—that is, the number of arguments that it takes.

A convenient way to bypass this is to assume that each function takes only one argument. In the case of an n-argument function, the return value after consuming the first argument is a function of $n-1$ arguments.

For instance, we rewrite the type of `max` as

```
max :: Int -> (Int -> Int)
```

This says that max takes as argument an Int and returns a function Int->Int. If we write

```
max 2 3
```

then (max 2) returns a function that we may think of as "max_of_2_and" where

```
max_of_2_and :: Int -> Int

max_of_2_and n = n, if n > 2
                 2, otherwise
```

In other words, we take a two argument function, freeze one argument and generate a new one argument function that reads only the second argument and has the frozen first argument "built in" to the definition.

To take another example, the function plus that adds two integers would now become

```
plus :: Int -> (Int -> Int)
```

and plus 2 3 would work as follows. (plus 2) returns a function "plus_2" where

```
plus_2 :: Int -> Int

plus_2 n = 2+n
```

Given this convention, we write a function f(x1,x2,...,xn) as f x1 x2 ... xn. This expression is implicitly bracketed from the left, so we have

```
f x1 x2 ... xn = (((f x1) x2) ... xn)
```

Let T1, T2, ..., Tn be the types of the n arguments of f and let T be the type of the return value. Then, rather than writing the type of f as (T1 x T2 x ... x Tn) -> T, we write

```
f :: T1 -> T2 -> ... -> Tn -> T
```

To be consistent with the implicit left bracketing of f x1 .. xn, we have to introduce brackets in the type expression from the *right*, as follows:

```
f :: T1 -> (T2 -> ... -> (Tn -> T))
```

This convention by which all functions are regarding as functions of a single argument is called *currying*, after the logician Haskell Curry, who has also lent his name to the Haskell language.[1]

[1] However, this idea was actually proposed by another logician called Schönfinkel.

11.2 Defining functions in Haskell

The most basic way of defining a function in Haskell is to "declare" what it does. For example, we can write:

```
double :: Int -> Int
double n = 2*n
```

Here, the first line specifies the type of the function and the second line tells us how the output of `double` depends on its input. We shall see soon that the "definition" of `double` is computed by treating the equality as a rule for rewriting expressions. We shall also see that, in Haskell, the type of a function can be inferred automatically, so we can omit the type when defining a function.

We are not restricted to having single line definitions for functions. We can use multiple definitions combined with implicit pattern matching. For instance consider the function:

```
power :: Float -> Int -> Float
power x 0 = 1.0
power x n =  x * (power x (n-1))
```

Here, the first equation is used if the second argument to `power` is 0. If the second argument is not 0, the first definition does not "match", so we proceed to the second definition. When multiple definitions are provided, they are scanned in order from top to bottom.

Here is another example of a function specified via multiple definitions, using pattern matching.

```
xor :: Bool -> Bool -> Bool
xor True  True  = False
xor False False = False
xor x     y     = True
```

Here, the first two lines explicitly describe two interesting patterns and the last line catches all combinations that do not match.

Another way to provide multiple definitions is to use conditional guards. For example:

```
max :: Int -> Int -> Int
max i j  | (i >= j) = i
         | (i < j)  = j
```

In this definition the vertical bar indicates a choice of definitions and each definition is preceded by a boolean condition that must be satisfied for that line to have effect. The boolean guards need not be exhaustive or mutually exclusive. If no guards are true, none of the definitions are used. If more than one guard is true, the earliest one is used.

It is important to note that all variables used in patterns are substituted independently—we cannot directly "match" arguments in the pattern by using the same variable for two arguments to implicitly check that they are the same. For instance, the following will not work.

```
isequal :: Int -> Int -> Bool
isequal x x = True
isequal y z = False
```

Instead, we must write

```
isequal :: Int -> Int -> Bool
isequal y z | (y == z) = True
            | (y /= z) = False
```

or, more succinctly,

```
isequal :: Int -> Int -> Bool
isequal y z = (y == z)
```

When using conditional guards, the special guard `otherwise` can be used as a default value if all other guards fail, as shown in the following example.

```
max3 i j k | (i >= j) && (i >= k) = i
           | (j >= k)             = j
           |  otherwise           = k
```

Basic types can be combined into n-tuples—for instance:

```
-- (x,y) :: (Float,Float) represents a point in 2D
distance :: (Float,Float) -> (Float,Float) -> Float
distance (x1,y1) (x2,y2) = sqrt((x2-x1)*(x2-x1) +
                                (y2-y1)*(y2-y1))
```

The first line in the definition above is a comment. Observe the use of pattern matching to directly extract the components of the tuples being passed to `distance`.

Auxiliary functions can be locally defined using `let` or `where`. The two notations are quite similar in effect, except that `let` can be nested while `where` is only allowed at the top level of a function definition. Formally, `let` forms part of the syntax of Haskell expressions while `where` is part of the syntax of function declarations. A definition using `where` can be used across multiple guarded options. Each choice in a guarded set of options is an independent Haskell expression and a definition using `let` is restricted to the expression in which it occurs, so we have to use a separate `let` for each guarded clause.

```
distance (x1,y1) (x2,y2) =
  let xdistance = x2 - x1
      ydistance = y2 - y1
      sqr z = z*z in
  in sqrt((sqr xdistance) + (sqr ydistance))
```

```
distance (x1,y1) (x2,y2) = sqrt((sqr xdistance) + (sqr ydistance))
where
  xdistance = x2 - x1
  ydistance = y2 - y1
  sqr z = z*z
```

11.3 Functions as first class objects

As we have seen, because of currying, a function typically has a return value that is itself a function (i.e., the value returned has a type of the form `X -> Y`, where `X` and `Y` may themselves be further function types). Symmetrically, there is nothing to stop us from functions whose input type is a function type.

Thus, there is no distinction between basic types such as `Int`, `Bool`, etc and function types as far as input and output types of functions are concerned. In the literature, this property is often referred to as having "functions as first class objects". Contrast this to a language like C or Java where there is no natural way to pass a function as an argument or as a return value.

A function that takes another function as input is sometimes called a "higher order" function. Here is a simple higher order function:

```
apply :: (Float -> Int) -> Float -> Int
apply f y = (f y)
```

Thus, `apply` takes two arguments and applies its first argument to its second argument. Notice that the types of `f` and `y` have to be compatible to permit this.

11.4 Polymorphism

The `apply` function defined earlier specifically requires the first argument to be a function with type `Float -> Int` and, hence, the second argument to be of type `Float`.

It would be more desirable to be able to write a version of `apply` whose first argument has an arbitrary function type `T -> T'` and whose second argument is of a matching type `T`, so that `apply` itself has the type `(T -> T') -> T -> T'`.

This is an example of *polymorphism*. The word means "taking multiple forms" and refers to a function that operates similarly on multiple types. An even more basic example of a polymorphic function is the identity function that just returns its argument. This function should have type `T -> T`, for any type `T`.

Other examples of polymorphism are familiar functions on lists. For instance, `reverse` reorders a list of any type, so reverse has type `[T] -> [T]` for all types `T` (recall that for a type `T`, the list of that type has type `[T]`). Similarly, length returns the length of any list, so length should have type `[T] -> Int`.

This kind of polymorphism is sometimes referred to as "structural" polymorphism because of the way it is reflected in functions such as `reverse` and `length`, where the function operates on the structure of the collection rather than on the base type.

One must be careful to distinguish this kind of polymorphism from the ad hoc variety associated with overloading operators. For instance, in most programming languages, we write `+` to denote addition for both integers and floating point numbers. However, since the underlying representations used for the two kinds of numbers are completely different, we are actually using the same name (`+`, in this case) to designate functions that are computed in a different manner for different base types.

Another example of ad hoc polymorphism is that available in Java, where different methods can have the same name but different parameter types. Thus, for instance, the class `Arrays` seemingly supports a uniform method called `sort` to sort arrays of arbitrary scalar types, but this is just a collection of distinct functions with the same name—the correct one is picked depending on the type of argument passed to the function.

Coming back to Haskell, we can describe polymorphic types using type variables, typically denoted `a`, `b`, `c`, ... Thus we have the following type definitions for the functions discussed earlier.

```
identity :: a -> a
apply :: (a -> b) -> a -> b
length :: [a] -> Int
reverse :: [a] -> [a]
```

Type variables come with implicit universal quantification. Thus, for instance, we read the type of `apply` as "for every type `a` and every type `b`, `(a -> b) -> a -> b`". Type variables are uniformly substituted by "real" types to get a specific instance of the function. Thus, for instance, when we define a function

```
plus2 :: Int -> Int
plus2 n = 2+n
```

and then write

```
apply plus2 5
```

this first instantiates both the variables `a` and `b` to type `Int` and uses the concrete definition

```
apply :: (Int -> Int) -> Int -> Int
```

to do the job.

11.5 Conditional polymorphism

What would be the type of a sort function? The definition

```
quicksort :: [a] -> [a]
```

is not accurate, because we cannot sort *any* list. We need to have a way of unambiguously comparing items in the list. This is not possible for arbitrary types because, for instance, `a` could be a function type and it is not always possible to effectively compare functions.

Consdier a simpler example—`memberof` that checks whether the value supplied as its first argument appears in the list supplied as its second argument. This should be of type

```
memberof :: a -> [a] -> Bool
```

For this, we have to be able to check whether two elements of type `a` are equal. Even this is problematic for arbitrary types—again, what if `a` is a function type? Elementary computability theory tells us that checking whether two functions are equal (i.e., agree on all inputs) is undecidable.

To get around this, we have to place restrictions on the type of `a`. In Haskell, this is done by organizing the collection of all types into type classes (nothing to do with classes as in object-oriented programming!) A type class in Haskell is just a subset of all possible types. For instance, the type class `Eq` is the set of all types for which equality checking is possible. We can then write the type of `memberof` as

```
memberof :: Eq a => a -> [a] -> Bool
```

This is to be read as "`memberof` is of type `a -> [a] -> Bool` for any type `a` that belongs to the class `Eq`".

In Haskell, the type class `Ord` contains all types on which comparisons are possible. So, we would have

```
quicksort :: Ord a => [a] -> [a]
```

Since comparing involves checking equality as well, clearly every type in `Ord` is also in `Eq`.

Recall that in Java, conditional polymorphism was achieved using interfaces. Unconditional polymorphism, at least for functions that operate on objects, can be achieved by defining the argument to be of type `Object`, which is compatible with all other classes in Java by the definition of the type hierarchy.

11.6 User defined datatypes

Haskell permits user defined datatypes. We can define structured types such as trees, stacks, queues .... To do this, we define new names, called *constructors*, to denote the components of the new type. In general, a constructor takes arguments. The simplest type of constructor is a constant, one that takes 0 arguments.

The simplest user defined datatype is an enumerated type, such as

```
data day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

More interesting are recursive types, such as a binary tree where each node can hold an integer.

```
data Btreeint = Nil | Node Int Btreeint Btreeint
```

This says that a binary tree of integers, Btreeint, is either an empty tree, Nil, or a node which holds an integer value and contains two nested trees, corresponding to the left and right children.

Can we define polymorphic datatypes, like polymorphic functions? For instance, can we define just the structure of a binary tree without fixing the type of value stored at each node?

Yes, we can, using the notion of a type variable, as before.

```
data Btree a = Nil | Node a (Btree a) (Btree a)
```

Note here that the complete name of the new type is Btree a not just Btree.

To take another example, we can redefine the builtin type list as follows:

```
data List a = Nil | Cons a (List a)
```

The next step is to declare that a user defined datatype belongs to a type class such as Eq or Ord. Each type class is characterized by a set of functions that each member of the class must support—analogous to an interface in Java. For instance, for a type to belong to Eq, it must support an equality check, denoted ==. The exact implementation of == is irrelevant. For instance, we might define == on trees to be isomorphism and place Btreeint in Eq as follows.

```
instance Eq Btreeint where
  Nil ==  Nil                          =  True
  (Node x t1 t2) == (Node y u1 u2) = (x == y) && (t1 == u1)
                                              && (t2 == u2)
```

The definition of == for Btreeint follows the normal syntax of a Haskell function definition—in this example, by cases.

How about making Btree a an element of Eq? We could analogously write

```
instance Eq (Btree a) where
  Nil ==  Nil                      =  True
  (Node x t1 t2) == (Node y u1 u2) = (x == y) && (t1 == u1)
                                              && (t2 == u2)
```

Notice however, that this uses the test `x == y` for elements of the unspecified type `a`. In `Btreeint` this was not a problem because the type of `x` and `y` was known to be `Int`, which belongs to `Eq`. Here, instead, we must make the membership of `Btree a` in `Eq` conditional on the properties of the underlying type `a`, as follows.

```
instance Eq a => Eq (Btree a) where
  Nil ==  Nil                      =  True
  (Node x t1 t2) == (Node y u1 u2) = (x == y) && (t1 == u1)
                                              && (t2 == u2)
```

In other words, "if `a` is in `Eq`, then `Btree a` is in `Eq` with `==` defined as . . . "

We have to be a bit careful here. With the definitions we have given, the Haskell interpreter responds as follows,

```
Main> (Node 5 Nil Nil) == (Node 6 Nil Nil)
False
Main> (Node 5 Nil Nil) == (Node 5 Nil Nil)
True
```

which is as we expect, but, somewhat unexpectedly, it says

```
Main> Nil == Nil
ERROR - Unresolved overloading
*** Type       : Eq a => Bool
*** Expression : Nil == Nil
```

This is because if we just say `Nil`, the interpreter cannot instantiate `a` to a concrete type and determine whether it belongs to `Eq` or not. The same happens if we try to ask whether `[] == []` for the empty list that is built in to Haskell. To get around this, we have to explicitly supply information about the base type of the tree or list as follows.

```
Main> (Nil::Btree Int) == Nil
True
Main> ([]::[Float]) == []
False
```

Notice that it is sufficient to specify the base type for `[]` on one side of the equation. Interestingly, even if we try, we cannot assert that there are two different instances of `[]` in an expression. For instance, if we try the following

```
Prelude> ([]::[Float]) == ([]::[Int])
```

the response is

```
ERROR - Type error in application
*** Expression     : [] == []
*** Term           : []
*** Type           : [Float]
*** Does not match : [Int]
```

11.7 Rewriting as a model of computation

The model of computation that underlies Haskell is rewriting. In a sense, rewriting is at the heart of all formulations of computability—any computational device's fundamental behaviour is to read symbols and generate other symbols, which amounts to rewriting the input as the output.

Function definitions in Haskell provide the rules for rewriting. Normally, rewriting reduces the complexity of an expression, hence *reduction* is a synonym for rewriting. Here, complexity is a semantic notion associated with the expression, not a syntactic notion. The expression that results after "reduction" is often longer and more complex, syntactically, than the orignal expression! In the literature, the term *rewriting system* is usually used to signify a set of rules that may be applied in either direction, while *reduction* refers to a rewriting process that uses the rules in one direction only.

For instance, given the function definition

```
  square: Int -> Int
  square n = n*n
```

we have square 7 ⇝7*7 ⇝49, where the symbol ⇝ denotes "rewrites to".

The first reduction is user specified, the second is built in.

Reduction continues till a *result* is obtained. A result is just an expression that cannot be rewritten further. In Haskell, results are normally constants that look sensible—values of type Int, Float, ... However, as we shall see in the lambda calculus, this may not always be the case—we may have complicated expressions that cannot be further reduced.

What is the reduction sequence for square (4+3)? We have more than one possibility:

- square (4+3) ⇝square 7 ⇝7*7 ⇝49

- square (4+3) ⇝(4+3) * (4+3) ⇝7 * (4+3) ⇝7*7 ⇝49

- square (4+3) ⇝(4+3) * (4+3) ⇝(4+3) * 7 ⇝7*7 ⇝49

Is it a coincidence that all three sequences yield the same result? No. In fact, we have the following theorem.

Theorem Let e be a Haskell expression with two reductions $e \leadsto f_1 \leadsto f_2 \leadsto \cdots \leadsto f_k \leadsto g$ and $e \leadsto f_1' \leadsto f_2' \leadsto \cdots \leadsto f_\ell' \leadsto g'$ such that g and g' cannot be further reduced. Then, $g = g'$.

In general, we need a *strategy* for reduction—that is, a rule telling us how to apply the rules! Some questions we might ask are:

1. Are all reduction strategies equivalent with respect to the final result (this property is called *confluence*)?

2. Can we tell whether the reduction process will terminate? For example, consider what happens to the expression `factorial -1` with respect to the definition

```
    factorial:: Int -> Int
    factorial n = n * (factorial (n-1))
```

3. Given a choice of reduction strategies, can we always identify the optimal one, in terms of efficiency? Notice in the reduction of `square (4+3)` that the first reduction sequence is shorter than the other two.

11.8 Reduction strategies

We can associate a notion of nesting with Haskell expressions. For instance, if we have something like `f (g x)`, then the expression `(g x)` is nested deeper than the expression `f`. Thus it makes sense to talk of outer and inner expressions as candidates for reduction.

Two canonical reduction strategies are

- Outermost = lazy = do not evaluate arguments till you need to

- Innermost = eager = evaluate arguments before evaluating function

The first reduction of `square (4+3)` above was computed eagerly, while the second and third were lazy. Haskell follows the strategy of outermost, or lazy, reduction, while ML follows the strategy of innermost, or eager, reduction.

Here is another example:

```
power :: Float -> Int -> Float
power x n  | (n==0)    = 1
           | otherwise = x * (power x (n-1))
```

Consider what happens when we evaluate `power (7.0/0) 0`.

With outermost reduction, we first try to reduce `power`. The first rule applies and we get `power (7.0/0) 0` $\leadsto$`1`. On the other hand, innermost reduction will try to evaluate `(7.0/0)` and generate an error.

Thus, the two reduction strategies are not always equivalent. We have the following general result.

Theorem The outermost strategy for reduction terminates with a value whenever the innermost strategy does.

The example of `power` shows that there are expressions where the innermost strategy does not terminate with a value while the outermost strategy does. We should not assume that "values" are always sensible. It is debatable, mathematically, whether `power (7.0/0) 0` should evaluate to `1`. All we are discussing is the relative power of the two reduction strategies.

The reduction we saw for `square (4+3)` suggests that outermost reduction involves wasteful recomputation—in this case, `(4+3)` is evaluated twice. In fact, Haskell uses an optimization of outermost reduction called *graph reduction*. Graph reduction maintains only one copy of each unreduced expression and a pointer to this copy from each occurrence in the main expression. Thus, if the same subexpression occurs multiple times in an expression, the expression will have multiple pointers to the location containing the subexpression. The first time this subexpression is encountered, it will be reduced. Whenever the subexpression is encountered subsequently, the pointer will reveal that it has already been reduced.

Theorem Outermost graph reduction is at least as efficient as innermost reduction.

In fact, in some situations outermost reduction can be significantly more efficient, because it will terminate without performing unnecessary computations. Consider the following definition of a binary tree which stores values only at the leaves.

```
data BTree a = Nil | Leaf a | Node BTree a BTree a
```

We would like to compare the list of values stored at the frontiers of such trees. Let us define a function that checks whether these lists are the same for two trees.

```
equaltree :: BTree a -> BTree a -> Bool
equaltree t1 t2 = ((leaflist t1) ==  (leaflist t2))

leaflist  :: BTree a -> [a]
leaflist Nil          = []
leaflist (Leaf x)     = [x]
leaflist (Node t1 t2) = (leaflist t1) ++ (leaflist t2)
```

Checking list equality in Haskell is lazy and terminates when the first pair of nonequal elements is encountered, so the Haskell version of this function will actually construct `leaflist t1` and `leaflist t2` only to the extent required to show that they are not the same. Of course, if they are the same, both lists must be constructed in entirety. In an eager functional language (like ML), a similar function would first compute `leaflist t1` and `leaflist t2` in entirety and only then check them for equality, which is more expensive in general.

11.9 Outermost reduction and infinite data structures

Outermost reduction permits the definition of infinite data structures. For instance, the list of all integers starting at n is given by the function

```
listfrom n = n: (listfrom (n+1))
```

The output of listfrom m is the infinite list [m, m+1, ... which is denoted [m..] in Haskell.

We can use infinite lists, for instance, to define in a natural way the Sieve of Eratosthenes whose output is the (infinite) list of all prime numbers.

```
sieve (x:xs) = x : (sieve [ y <- xs | mod y x != 0])
primes = sieve [2..]
```

The function sieve picks up the first number of its input list, constructs a new list from its input by removing all multiples of the first number and then recursively runs sieve on this list.

If we work out the reduction for this we get

```
primes ⇝  sieve [2..]
       ⇝  2 :  (sieve [ y | y <- [3..]  , mod y 2 != 0])
       ⇝  2 :  (sieve (3 :  [y | y <- [4..], mod y 2 != 0])
       ⇝  2 :  (3 :
            (sieve [z  | z <- (sieve [y | y <- [4..], mod y 2 != 0])
                           | mod z 3 != 0])
       ⇝  2 :  (3 :  (5 :
            (sieve [w | w <- (sieve [z | z <- (sieve [y | y <- [4..],
               mod y 2 != 0]) | mod z 3 != 0]) | mod w 5 != 0])
       ⇝  ...
```

Why is this useful? It is often conceptually easier to define a function that returns an infinite list and extract a finite prefix to get a concrete value.

For instance, from primes we can derive functions such as

```
nthprime k = primes !! k
```

that extracts the kth prime number.

Chapter 12

The (untyped) lambda calculus

The lambda calculus was invented by Alonzo Church around 1930 as an attempt to formalize a notation for computable functions. It is important to have some basic familiarity with the lambda calculus because it forms the foundation for the definition of functional programming languages such as Haskell.

In the conventional set theoretic formulation of mathematics, a function f can be represented by its graph—that is, a binary relation R_f between the sets $\mathit{domain}(f)$ and $\mathit{codomain}(f)$ consisting of all pairs $(x, f(x))$. Not all binary relations represent functions—functions are single-valued, so if $(x, y) \in R_f$ and $(x, y') \in R_f$ then we must have $y = y'$. In mathematics, functions are almost always total, so we also have an additional requirement that for each x in $\mathit{domain}(f)$, there exists a pair $(x, y) \in R_f$. Representing a function in terms of its graph is called an *extensional* definition—two functions are equal if and only if their graphs are equal. This is analogous to the definition of equality in set theory—two sets are equal if and only if they have the same elements.

From a computational viewpoint, it is not sensible to represent functions merely in terms of graphs. Clearly, all sorting algorithms define functions that have the same graph. However, from a computational viewpoint, we would like to distinguish efficient sorting algorithms from inefficient ones.

What we need is an *intensional* representation of functions—a representation that captures not just the output value for each input, but also how this value is arrived at.

With this background, we plunge into the (pure untyped) lambda calculus.

12.1 Syntax

The set of lambda expressions is generated by what we would, in modern day computer science terminology, call a context-free grammar.

Let *Var* be a countably infinite set of variables. We define the set Λ of lambda expressions as follows:

$$\Lambda = x \mid \lambda x.M \mid MM'$$

where $x \in Var$ and $M, M' \in \Lambda$.

Thus, the syntax of lambda expressions contains two ways to generate new expressions from old ones. The construction $\lambda x.M$ is called *abstraction*, while the construction MM' is called *application*.

The intention is that $\lambda x.M$ is an expression representing a function of x whose body (or rule for computation) is given by M. Thus, $\lambda x.M$ "abstracts" the computation rule described in M to operate on arbitrary input values x. This is analogous to writing $f(x) = M$ without having to assign an unnecessary "name" f to the function (after all, $f(x) = M$ and $g(x) = M$ are clearly the same function, so the names f and g carry no significance.)

The expression MM' is intended to represent the action of "applying" the function encoded by M to the argument M'.

12.2 Life without types

At this point, we should remind ourselves that these intended meanings do not stop us from constructing seemingly nonsensical expressions such as xx.

At first sight, it seems strange that we should try to embark on a syntax for functions in which there is no notion of "type". However, untyped formalisms are familiar to us

- Set theory is untyped. Everything is a set. To do arithmetic, we define certain sets to denote numbers. Functions on numbers apply in principle to all sets but are "meaningless" in the wrong context.

- A computer's memory is intrinsically untyped. Consider the operation of *dereferencing a pointer* in C—that is, treat a memory location as a pointer and look up the location that it points to. This operation is performed without checking if the contents of the pointer are of the right "type"—that is, a memory location legally available to this program. If the contents of the pointer are valid, the operation succeeds. However, if the pointer is invalid (for instance, if it is uninitialized), the operation fails (sometimes dramatically, as we all know!)

The point is that we have a syntax for representing data (sets, bit strings) and we have rules for manipulating these representations. Some representations are more meaningful than others, but the manipulation rules can be applied uniformly to both the meaningful and the meaningless representations. All we ask is that if the original data represents a meaningful value, then, after manipulation, it still represents a meaningful value.

A similar view is taken in the lambda calculus—for instance MM' represents application if the structure of the expression M corresponds to a "function". Otherwise, we don't care what this construction means (e.g., if MM' is xx, as describe earlier).

12.3 The rule β

The basic rule for computing with the lambda calculus is called β and is given by:

$$(\lambda x.M)M' \rightarrow_\beta M\{x \leftarrow M'\}$$

where $M\{x \leftarrow M'\}$ represents the effect of substituting all *free* occurrences of x in M uniformly by M'. (We shall formally define what we mean by a free occurrence of x shortly.) The β rule is one that we use unconciously all the time: If $f(x) = 2x^2 + 3x + 4$, then $f(7) = 2 \cdot 7^2 + 3 \cdot 7 + 4$ which is just $2x^2 + 3x + 4\{x \leftarrow 7\}$.

From the β rule it is clear that an application MM' is meaningful only if M is of the form $\lambda x.M$". Thus, though we can write terms like xx, we cannot do much with them.

Effectively, the β rule is the *only* rule we need to set up the lambda calculus. For the moment, think of it as a rewriting rule for expressions that can be applied to any subterm in a lambda expression. We will formalize this later.

12.4 Variable capture

An important issue to keep in mind when applying the rule β is that the substitution for x should not "capture" variables. Consider, for instance, the application $(\lambda x.(\lambda y.xy))y$. Using the β rule naively, we get $(\lambda x.(\lambda y.xy))y \rightarrow_\beta \lambda y.yy$ However, the y that comes in to $\lambda y.xy$ by the substitution for x is a *new* y and should not be confused with the y that is used to define the inner function. Thus, we should redefine the inner function with a fresh variable before making the substitution, as follows:

$$(\lambda x.(\lambda y.xy))y = (\lambda x.(\lambda z.xz))y \rightarrow_\beta \lambda z.yz$$

Clearly, changing the name of the variable defining a function does not affect the definition—there is no difference between the functions $f(x) = 2x + 5$ and $f(z) = 2z + 5$.

To formalize this notion we have to define *free* and *bound* variables. Intuitively, a variable is free in an expression until it is "bound" by an abstraction. Thus, in the expression xy, both x and y are free. If we abstract y as $\lambda y.xy$, x remains free but y becomes bound. More formally, we can define two sets $FV(M)$ (*free variables of* M) and $BV(M)$ (*bound variables of* M) inductively, as follows:

- $FV(x) = \{x\}$, for any variable x
- $FV(\lambda x.M) = FV(M) - \{x\}$
- $FV(MM') = FV(M) \cup FV(M')$

- $BV(x) = \emptyset$, for any variable x
- $BV(\lambda x.M) = BV(M) \cup \{x\}$
- $BV(MM') = BV(M) \cup BV(M')$

When we apply the β rule to transform $\lambda x.MM'$ into $M\{x \leftarrow M'\}$, we need to check that no free variable in M' clashes with a bound variable in M. In the example above,

$$(\lambda x.(\lambda y.xy))y \rightarrow_\beta \lambda y.yy,$$

y is bound in the expression $M = \lambda y.xy$, while y is free in the expression $M' = y$.

In the literature, several approaches have been proposed to deal with the variable renaming problem that is required for the β rule to work properly. For simplicity, we shall just assume that we always rename the bound variables in M to avoid "capturing" free variables from M'.

12.5 Encoding arithmetic

Having set up this rather abstract notation for computable functions, let us see an example of how to use it.

Before proceeding, we set up some conventions. Implicitly, lambda expressions associate to the left. Thus, the expresssion $M_1M_2M_3$ denotes $((M_1M_2)M_3)$. Also, if we have an abstraction with multiple variables at the outermost level, such as $\lambda x.(\lambda y.xy)$, we can pull out all the variables at the outermost level into a single sequence such as $\lambda xy.xy$.

In set theory, the number n is identified, in a sense, with a set that has nesting level of depth n. For instance, we start with the emptyset for representing the number 0 and, inductively, use the set $\{0, 1, .., n-1\}$ to represent the number n. Thus we have

$$\begin{array}{rcl}
0 & = & \emptyset \\
1 & = & \{\emptyset\} \\
2 & = & \{\emptyset, \{\emptyset\}\} \\
3 & = & \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\
\ldots & &
\end{array}$$

In the lambda calculus, we encode the number n by the number of times we apply a function to an argument. Thus, the number n is a lambda expression that takes two arguments and applies the first argument to the second argument n times. Let $\langle n\rangle$ be an abbreviation for the lambda term denoting n. Then, we have:

$$\begin{array}{rcl}
\langle 0\rangle & = & \lambda fx.x \\
\langle n+1\rangle & = & \lambda fx.f(\langle n\rangle fx)
\end{array}$$

This encoding was proposed by Church and terms of the form $\langle n\rangle$ are hence called the *Church numerals*. Concretely, we have

$$\langle 1\rangle = \lambda fx.f(\langle 0\rangle fx) = \lambda fx.(f((\lambda fx.x)fx))$$

Note that $\langle 0\rangle gy \rightarrow_\beta (\lambda x.x)y \rightarrow_\beta y$. Thus $\langle 0\rangle fx \rightarrow_\beta x$. So, inside the definition of $\langle 1\rangle$, we get

$$\langle 1\rangle = \ldots = \lambda fx.(f\,\underbrace{\lambda fx.x)fx)}_{\text{apply }\beta})\rightarrow_\beta \lambda fx.(fx)$$

Since $\langle 1\rangle gy \rightarrow_\beta (\lambda x.(gx))y \rightarrow_\beta gy$, we have

$$\langle 2\rangle = \lambda fx.f(\langle 1\rangle fx) = \lambda fx.(f(\underbrace{\lambda fx.(fx)fx)}_{\text{apply } \beta}) \rightarrow_\beta \lambda fx.(f(fx))$$

and

$$\langle 2\rangle gy \rightarrow_\beta \lambda x.(g(gx))y = g(gy)$$

Let us write $g^k y$ to denote the term $g(g(\ldots(gy)))$ with k applications of g to y. Then, it is not difficult to show inductively that

$$\langle n\rangle = \lambda fx.f(\langle n{-}1\rangle fx) \rightarrow_\beta \ldots \rightarrow_\beta \lambda fx.(f^n x)$$

We can now define arithmetic functions such as *successor*, *plus*, ... For instance, *successor* is just $\lambda pfx.f(pfx)$. To see how this works, we observe that

$$(\lambda pfx.f(pfx))\langle n\rangle \rightarrow_\beta \lambda fx.f(\langle n\rangle fx) \rightarrow_\beta \lambda fx.f(f^n x) = \lambda fx.f^{n+1}x = \langle n{+}1\rangle$$

Thus, if the function representing successor is applied a term of the right "type", it yields the expected answer. We could also apply it to a "meaningless" term, in which case we get a "meaningless" answer!

Here is a definition for *plus*: $\lambda pqfx.pf(qfx)$. We can verify that

$$\begin{array}{rcl}
(\lambda pqfx.pf(qfx))\langle m\rangle\langle n\rangle & \rightarrow_\beta & (\lambda qfx.\langle m\rangle f(qfx))\langle n\rangle \\
& \rightarrow_\beta & (\lambda fx.\langle m\rangle f(\langle n\rangle fx)) \\
& \rightarrow_\beta & (\lambda fx.\langle m\rangle f(f^n x)) \\
& \rightarrow_\beta & (\lambda fx.f^m(f^n x)) \\
& = & (\lambda fx.f^{m+n}x) = \langle m{+}n\rangle
\end{array}$$

Similarly, we define *multiplication* and *exponentiation* as follows:

$$\begin{array}{lcl}
\textit{multiplication} & : & \lambda pqfx.q(pf)x \\
\textit{exponentiation} & : & \lambda pq.(pq)
\end{array}$$

12.6 One step reduction

Recall that we introduced the β rule

$$(\lambda x.M)M' \rightarrow_\beta M\{x \leftarrow M'\}$$

and said, informally, that we would permit this rule to be used in all "contexts". The β rule is not the only basic rule possible. For instance, observe that the two terms $\lambda x.(Mx)$ and M are equivalent with respect to β reduction—for any term M', $(\lambda x.(Mx))M' \rightarrow_\beta MM'$. This can be formalized by a rule (normally called η) which says

$$\lambda x.(Mx) \rightarrow_\eta M$$

Given a set of *basic* rules such as β, η, ..., we can inductively define a one step reduction that permits any of these basic rules to be used in any context. Let us denote one step reduction by $\rightarrow$. We define $\rightarrow$ through inference rules such as the following:

$$\text{if } M \rightarrow_x M' \text{ for some basic rule } x \text{ such as } \beta,\ \eta,\ \ldots, \text{ then } M \rightarrow M'$$

Following the conventional notation used in logic, we present such a rule in the following form

$$\frac{M \rightarrow_x M'}{M \rightarrow M'}$$

Here is the complete set of rules defining $\rightarrow$:

$$\frac{M \rightarrow_x M''}{M \rightarrow M} \qquad \frac{M \rightarrow M'}{\lambda x.M \rightarrow \lambda x.M'} \qquad \frac{M \rightarrow M'}{MN \rightarrow M'N} \qquad \frac{N \rightarrow N'}{MN \rightarrow MN'}$$

Notice that all that we have done is to formalize the fact that we permit the basic reductions $\rightarrow_x$ within any subterm. In the "calculations" we have seen so far, we have already informally used this form of applying the β rule.

In the discussion that follows, we shall dispense with the η rule and assume that the only basic rule is the rule β.

12.7 Normal forms

As we had discussed in the context of Haskell, we are interested in terms that cannot be reduced. These denote "values" and are called *normal forms*.

Formally, we are interested in properties of the relation $\rightarrow^*$, the reflexive and transitive closure of $\rightarrow$. We have already anticipated some of the questions we would like to ask about $\rightarrow^*$ in our discussion of Haskell.

- Does every term reduce to a normal form?
- Can a term reduce to more than one normal form, depending on the order in which we perform reductions?
- If a term has a normal form, can we always find it?

Consider the term $(\lambda x.xx)(\lambda x.xx)$. Only one reduction step is possible and this makes two copies of the argument and returns us to the original term $(\lambda x.xx)(\lambda x.xx)$. The reduction process for this term never terminates, so this is an example of a lambda term without a normal form. In the literature, the term $(\lambda x.xx)(\lambda x.xx)$ is usually denoted Ω.

Consider now the term $(\lambda yz.z)$ that ignores its first argument. If we apply $(\lambda yz.z)$ to Ω, we have two possible reductions.

- $(\lambda yz.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow \lambda z.z$, using outermost reduction.

- $(\lambda yz.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow (\lambda yz.z)((\lambda x.xx)(\lambda x.xx))$, using innermost reduction.

Clearly, if we keep making the second choice, we will never reach a normal form, even though it is always possible to reach a normal form in one step by choosing the first reduction.

To answer the question of multiple normal forms, we will state a more general result shortly.

First, we define a notion of equivalence based on $\rightarrow^*$.

12.8 $\rightarrow^*$-equivalence

We define an equivalence on lambda terms, which we shall write $\leftrightarrow$, based on the many step reduction relation $\rightarrow^*$. Basically, $\leftrightarrow$ is the symmetric transitive closure of $\rightarrow^*$. It can be defined inductively using inference rules in the same way that we defined $\rightarrow$ based on $\rightarrow_x$:

$$\frac{M \rightarrow^* N}{M \leftrightarrow N} \quad \frac{M \leftrightarrow N}{N \leftrightarrow M} \quad \frac{M \leftrightarrow N, N \leftrightarrow P}{M \leftrightarrow P}$$

We can do the same with any reflexive, transitive relation R— the symmetric, transitive closure of R defines an equivalence relation $\overset{R}{\leftrightarrow}$.

12.9 Church-Rosser property

Let R be a binary relation that is reflexive and transitive. We say that R has the *diamond property* if, whenever $X\ R\ Y$ and $X\ R\ Z$, then there is a W such that $Y\ R\ W$ and $Z\ R\ W$. If a relation R has the diamond property, we say that R is Church-Rosser.

Theorem *(Church-Rosser)* Let R be Church-Rosser. Then $M \overset{R}{\leftrightarrow} N$ implies there exists Z, $M\ R\ Z$ and $N\ R\ Z$

Proof By induction on the definition of $\overset{R}{\leftrightarrow}$.

(i) $M \overset{R}{\leftrightarrow} N$ because $M\ R\ N$. Then chooze $Z \equiv N$.

(ii) $M \overset{R}{\leftrightarrow} N$ because $N \overset{R}{\leftrightarrow} M$. Trivial.

(iii) $M \overset{R}{\leftrightarrow} N$ because $M \overset{R}{\leftrightarrow} P$ and $P \overset{R}{\leftrightarrow} N$. By the induction hypothesis, we have Y such that $M\ R\ Y$ and $P\ R\ Y$ and Z such that $P\ R\ Z$ and $N\ R\ Z$.

Since $P\ R\ Y$, $P\ R\ Z$ and R is Church-Rosser, there is a term W such that $Y\ R\ W$ and $Z\ R\ W$.

But then $M\ R\ W$ and $N\ R\ W$.

□

We have the following useful Corollary:

Corollary Let R be Church-Rosser. Then a term can have at most one R-normal form (where an R-normal form for M is, as expected, a term N such that $M\ R^*\ N$ and there is no P such that $N\ R\ P$.)

Proof Suppose N_1 and N_2 are both normal forms for M. Then, $N_1 \overset{R}{\leftrightarrow} M \overset{R}{\leftrightarrow} N_2$, so $N_1 \overset{R}{\leftrightarrow} N_2$. There must then be a common Z such that $N_1\ R\ Z$ and $N_2\ R\ Z$. Since neither N_1 nor N_2 can be reduced, it follows that $N1 \equiv N_2 \equiv Z$. □

Recall that our goal is to check whether normal forms in the lambda calculus are unique. One way to do this is to show that $\rightarrow^*$ has the Church-Rosser property.

For any relation $\Rightarrow$, if $\Rightarrow$ has the Church-Rosser property, then so does $\Rightarrow^*$. The proof of this is easy to see pictorially.

$$\begin{tikzcd}
M \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & N_1 \arrow[d, Rightarrow] \\
{} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[d, phantom, "\vdots"] \\
{} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {} \arrow[r, Rightarrow] \arrow[d, phantom, "\vdots"] & {} & \cdots \\
N_2 \arrow[r, Rightarrow] & {} \arrow[r, Rightarrow] & {} & \cdots & &
\end{tikzcd}$$

If $M \Rightarrow^* N_1$ (the horizontal axis) and $M \Rightarrow^* N_2$ (along the vertical axis), we can break up both reductions into a sequence of individual steps. From the fact that $\Rightarrow$ is Church-Rosser, we can complete the diamond for each single step in the sequence. In the end, we get a complete rectangle between N_1 and N_2 and the corner opposite M is the common term that we can obtain from N_1 and N_2.

Unfortunately, $\rightarrow$ does not have the Church-Rosser property, so we cannot use this strategy to establish that $\rightarrow^*$ is Church-Rosser! Consider the term $(\lambda x.xx)((\lambda x.x)(\lambda x.x))$. We have two reductions possible:

- $(\lambda x.xx)((\lambda x.x)(\lambda x.x)) \rightarrow ((\lambda x.x)(\lambda x.x))((\lambda x.x)(\lambda x.x))$ (Outermost)
- $(\lambda x.xx)((\lambda x.x)(\lambda x.x)) \rightarrow ((\lambda x.xx)(\lambda x.x)$ (Innermost)

If we take the second option, we can then in one step get

$$(\lambda x.xx)(\lambda x.x) \rightarrow ((\lambda x.x)(\lambda x.x))$$

We can reach this term from the first option as well, but it requires *two* steps!

The solution lies in defining an alternative notion of one step reduction from the basic beta reduction $\rightarrow_\beta$ such that

(i) This new reduction is Church-Rosser.

(ii) Its reflexive, transitive closure is equal to $\rightarrow^*$.

We define this new reduction $\twoheadrightarrow$ as follows.

$$M \twoheadrightarrow M \qquad \frac{M \twoheadrightarrow M'}{\lambda x.M \twoheadrightarrow \lambda x.M'} \qquad \frac{M \twoheadrightarrow M', N \twoheadrightarrow N'}{MN \twoheadrightarrow M'N'} \qquad \frac{M \twoheadrightarrow M', N \twoheadrightarrow N'}{(\lambda x.M)N \twoheadrightarrow M'\{x \leftarrow N'\}}$$

The last inference rule builds in β reduction. The crux of the relation $\twoheadrightarrow$ is that it combines nonoverlapping $\rightarrow$ reductions in one parallel step. For instance, if both M and N can be reduced to M' and N', respectively, then MN can be reduced to $M'N'$ in one step.

We claim, without proof, that $\twoheadrightarrow$ is Church-Rosser and that $\twoheadrightarrow^*$ is the same as $\rightarrow^*$. Assuming the claim, it then follows that $\rightarrow^*$ is Church-Rosser and so the normal form for a term in the lambda calculus is unique, if it exists.

12.10 Computability

We now argue that all computable functions can be described in the lambda calculus. As is usual in computability theory, we restrict our attention to functions on natural numbers. Recall that we can encode natural numbers in the lambda calculus via the Church numerals $\langle n \rangle \equiv \lambda fx.f^n x$. Our goal now is to show that for every computable function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ we can find a lambda term $\langle f \rangle$ that encodes f so that for every k-tuple $n_1, n_2, \ldots, n_k$

$$\langle f \rangle \langle n_1 \rangle \langle n_2 \rangle \ldots \langle n_k \rangle \rightarrow^* \langle f(n_1, n_2, \ldots, n_k) \rangle$$

In other words, the result of applying the encoding of f to the encodings of the arguments n_1, n_2, …, n_k should be the encoding of the result $f(n_1, n_2, \ldots, n_k)$. Notice that on the left hand side we have used currying to break up the k-tuple of arguments to f into a sequence of k arguments to $\langle f \rangle$.

Recursive functions

One of the many formulations of computable functions that arose in the quest to define effective computability is that of *recursive functions*, due to Gödel. The recursive functions are defined inductively in terms of a set of initial functions and rules for combining functions, as follows.

Initial functions

- *Zero*: $Z(n) = 0$.
- *Successor*: $S(n) = n+1$.
- *Projection*: $\Pi_i^k(n_1, n_2, \ldots, n_k) = n_i$.[1]

[1] Notice that the projection functions are actually form an infinite family, one for each choice of k and $i \in \{1, 2, \ldots, k\}$.

Composition

Given $f : \mathbb{N}^k \to \mathbb{N}$ and $g_1, g_2, \ldots, g_k : \mathbb{N}^h \to \mathbb{N}$, the *composition* of f and $g_1, g_2, \ldots, g_k$, denoted $f \circ (g_1, g_2, \ldots, g_k)$, is the function

$$\begin{aligned}&f \circ (g_1, g_2, \ldots, g_k)(n_1, n_2, \ldots, n_h) = \\ &\qquad f(g_1(n_1, n_2, \ldots, n_h), g_2(n_1, n_2, \ldots, n_h), \ldots, g_k(n_1, n_2, \ldots, n_h))\end{aligned}$$

For example, the function $f(n) = n{+}2$ can be defined as the composition $f = S \circ S$ of the successor function with itself.

Primitive recursion

Given $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$, the function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is defined by *primitive recursion* from g and h if the following equalities hold:

$$\begin{array}{rcl} f(0, n_1, n_2, \ldots, n_k) & = & g(n_1, n_2, \ldots, n_k) \\ f(n{+}1, n_1, \ldots, n_k) & = & h(n, f(n, n_1, n_2, \ldots, n_k), n_1, \ldots, n_k) \end{array}$$

Here are some examples of primitive recursive definitions.

- The function $\mathit{plus}(n, m) = n{+}m$ can be defined by primitive recursion from $g = \Pi^1_1$ and $h = S \circ \Pi^3_2$. In other words,

$$\begin{array}{lclclcl} \mathit{plus}(0, n) & = & g(n) & = & \Pi^1_1(n) & = & n \\ \mathit{plus}(m{+}1, n) & = & h(m, \mathit{plus}(m, n), n) & = & S \circ \Pi^3_2(m, \mathit{plus}(m, n), n) & = & S(\mathit{plus}(m, n)) \end{array}$$

- The function $\mathit{times}(n, m) = n \cdot m$ can be defined by primitive recursion from $g = Z$ and $h = \mathit{plus} \circ (\Pi^3_3, \Pi^3_2)$.

Minimalization

Given $g : \mathbb{N}^{k+1} \to \mathbb{N}$, the function $f : \mathbb{N}^k \to \mathbb{N}$ is defined by *minimalization* from g if

$$f(n_1, n_2, \ldots, n_k) = \mu n.(g(n, n_1, n_2 \ldots, n_k) = 0)$$

where μ is the *minimalization operator*: $\mu n.P(n)$ returns the least natural number n such that $P(n)$ holds. If $P(n)$ does not hold for any n, then the result is undefined.

In modern algorithmic notation, f can be computed by a while loop of the form

```
n := 0;
while (g(n,n1,n2,...,nk) != 0) {n := n+1};
return n;
```

For example, consider the function $\log_2 n$ defined as follows: $\log_2 n = k$, if $2^k = n$. If n is not a power of 2, $\log_2 n$ is undefined. The function $\log_2$ can be defined by minimalization from the function $g(k, n) = 2^k - n$.[2]

[2] Technically, we have to be a bit more careful here. Since we are dealing only with functions over $\mathbb{N}$,

Recursive functions

The set of *recursive functions* is the smallest set which contains the initial functions and is closed with respect to composition, primitive recursion, and minimalization.

If we exclude minimalization from this construction, we get the set of *primitive recursive functions*, all of which are total functions. Minimalization is the only operator which introduces partial functions (or, from a computational standpoint, non-termination).

12.11 Encoding recursive functions in lambda calculus

Recall that we have already fixed the following definitions:

- $\langle n\rangle \equiv \lambda fx.(f^n x)$.
- $succ \equiv \lambda nfx.(f(nfx))$ such that $succ\langle n\rangle \rightarrow^* \langle n+1\rangle$.

The function *succ* is the encoding $\langle S\rangle$ for the initial function *Successor.* We can define encodings for the other initial functions *Zero* and *Projection* as follows:

- $\langle Z\rangle \equiv \lambda x.(\lambda gy.y)$.
- $\langle \Pi_i^k\rangle \equiv \lambda x_1 x_2 \ldots x_k.x_i$.

Similarly, the composition of functions is easily defined in the lambda calculus. What remains is to encode primitive recursion and minimalization. We begin by defining encodings for constructing pairs and projecting out the components of a pair.

- $pair \equiv \lambda xyz.(zxy)$.
- $fst \equiv \lambda p.(p(\lambda xy.x))$.
- $snd \equiv \lambda p.(p(\lambda xy.y))$.

Observe that $fst(pair\ a\ b) \equiv (\lambda p.(p(\lambda xy.x)))((\lambda xyz.(zxy))\ a\ b) \rightarrow (\lambda p.(p(\lambda xy.x))) \rightarrow (\lambda z.(zab))(\lambda xy.x) \rightarrow (\lambda xy.x)ab \rightarrow (\lambda y.a)b \rightarrow a$. Similarly, we can show that $snd(pair\ a\ b) \rightarrow^* b$.

We now show how to encode primitive recursion. For notational simplicity, we restrict ourselves to the case where the function f being defined by primitive recursion has just one argument. Thus, the definition of primitive recursion becomes

$$\begin{array}{lcl} f(0) & = & g \\ f(n{+}1) & = & h(n, f(n)) \end{array}$$

$m{-}n$ would be defined to be 0 if $m < n$, so we have to modify the definition of $g(k,n)$ appropriately to take care of this.

Observe that g is just a constant—that is, some fixed number k. Inductively assume that h is a function defined by the lambda expression $\langle h\rangle$—in other words, for all numbers m and n, $\langle h\rangle\langle m\rangle\langle n\rangle \rightarrow^* \langle g(m,n)\rangle$. The aim is to come up with a lambda expression $\langle f\rangle$ defining f, such that for all n, $\langle f\rangle\langle n\rangle \rightarrow^* \langle f(n)\rangle$.

Consider the function $t(n) = (n, f(n))$. Then, t can be defined by primitive recursion as follows:

$$\begin{array}{lclcl} t(0) & = & (0, f(0)) & = & (0, g) \\ t(n{+}1) & = & (n{+}1, f(n{+}1)) & = & (n{+}1, h(n, f(n))) \\ & & & = & (\mathit{succ}(\mathit{fst}(t(n))), h(\mathit{fst}(t(n)), \mathit{sec}(t(n)))) \end{array}$$

This is a simpler form of primitive recursion called *iteration*—think of it as repeatedly iterating a *step* function starting with $(0, g)$. (In programming language parlance, this is equivalent to translating a top-down recursive definition into an equivalent bottom-up iterative one).

The step function is lambda-definable by the following expression:

$$\mathit{step} \equiv \lambda x.\mathit{pair}(\mathit{succ}(\mathit{fst}\ x))(\langle h\rangle(\mathit{fst}\ x)(\mathit{sec}\ x)).$$

We can check that $\mathit{step}\ (\mathit{pair}\ \langle n\rangle\langle f(n)\rangle) \rightarrow^* \mathit{pair}\ \langle n{+}1\rangle\langle f(n{+}1)\rangle$.

$$\begin{array}{rcl} \mathit{step}\ (\mathit{pair}\ \langle n\rangle\ \langle f(n)\rangle) & \rightarrow & \mathit{pair}\ \ (\mathit{succ}(\mathit{fst}(\mathit{pair}\ \langle n\rangle\ \langle f(n)\rangle))) \\ & & \qquad (\langle h\rangle(\mathit{fst}(\mathit{pair}\ \langle n\rangle\ \langle f(n)\rangle))(\mathit{sec}(\mathit{pair}\langle n\rangle\ \langle f(n)\rangle))) \\ & \rightarrow & \mathit{pair}\ (\mathit{succ}\ \langle n\rangle)\ (\langle h\rangle\langle n\rangle\langle f(n)\rangle) \\ & \rightarrow & \mathit{pair}\ (\mathit{succ}\ \langle n\rangle)\ \langle h(n, f(n))\rangle \\ & \rightarrow & \mathit{pair}\ \langle n{+}1\rangle\ \langle h(n, f(n))\rangle \\ & \rightarrow & \mathit{pair}\ \langle n{+}1\rangle\ \langle f(n{+}1)\rangle. \end{array}$$

The encoding for t is as follows:

$$\langle t\rangle \equiv \lambda y.y\ \mathit{step}\ (\mathit{pair}\ \langle 0\rangle\ \langle g\rangle)$$

We can check that for all n, $\langle t\rangle\langle n\rangle \rightarrow^* \mathit{pair}\ \langle n\rangle\ \langle f(n)\rangle$.

$$\begin{array}{ll} \langle t\rangle\langle 0\rangle \rightarrow \langle 0\rangle\ \mathit{step}\ (\mathit{pair}\ \langle 0\rangle\ \langle g\rangle) \rightarrow \mathit{pair}\ \langle 0\rangle\ \langle g\rangle & \\ \langle t\rangle\langle n{+}1\rangle & \rightarrow \langle n{+}1\rangle\ \mathit{step}\ (\mathit{pair}\ \langle 0\rangle\ \langle g\rangle) \\ & \rightarrow \mathit{step}\ (\langle n\rangle\ \mathit{step}\ (\mathit{pair}\ \langle 0\rangle\ \langle g\rangle)) \\ & \rightarrow \mathit{step}\ (\mathit{pair}\ \langle n\rangle\ \langle f(n)\rangle)(\text{by ind. hyp.}) \\ & \rightarrow \mathit{pair}\ \langle n{+}1\rangle\ \langle f(n{+}1)\rangle(\text{by what has been proved above}) \end{array}$$

Since $f(n)$ is $\mathit{sec}(t(n))$, $\langle f\rangle$ is the following expression:

$$\langle f\rangle \equiv \lambda y.\mathit{sec}(\langle t\rangle y)$$

It is immediate that for all n, $\langle f\rangle\langle n\rangle \rightarrow^* \langle f(n)\rangle$.

Collapsing all the steps we have seen above, we can write down an expression PR that constructs a primitive recursive definition from the functions g and h:

$$PR \equiv \lambda hgy.sec(y(\lambda x.pair\ (succ\ (fst\ x))\ (h(fst\ x))\ (sec\ x)))(pair\ \langle 0\rangle\ g))$$

We could also have used recursive definitions of lambda experssions to encode primitive recursion. Recursive definitions can also be used to define minimalization. It turns out that every recursive definition in the lambda calculus can be "solved" by finding its fixed point.

12.12 Fixed points

We begin our discussion of fixed point by providing an encoding for boolean values true (`tt`) and false (`ff`) and conditional expressions. We begin with the conditional. We seek a term $\lambda bxy.\ E$ that will take three arguments, return x if b (the conditional expression) is true and y if b is false.

We claim that the expression $\lambda bxy.\ bxy$ with the definitions of `tt` and `ff` given by $\langle\texttt{tt}\rangle \equiv \lambda xy.x$ and $\langle\texttt{ff}\rangle \equiv \lambda xy.y$ do the job. (Incidentally, note that $\langle\texttt{ff}\rangle$ is the same as $\langle 0\rangle$).

Clearly

$$\begin{array}{rcl} (\lambda bxy.bxy)\langle\texttt{tt}\rangle fg & \rightarrow & \lambda xy.((\lambda xy.x)xy)fg \\ & \rightarrow & \lambda y.((\lambda xy.x)fy)g \\ & \rightarrow & (\lambda xy.x)fg \\ & \rightarrow & (\lambda y.f)g \\ & \rightarrow & f \end{array}$$

and, similarly,

$$\begin{array}{rcl} (\lambda bxy.bxy)\langle\texttt{ff}\rangle fg & \rightarrow & \lambda xy.((\lambda xy.y)xy)fg \\ & \rightarrow & \lambda y.((\lambda xy.y)fy)g \\ & \rightarrow & (\lambda xy.y)fg \\ & \rightarrow & (\lambda y.y)g \\ & \rightarrow & g \end{array}$$

Thus $\lambda bxy.bxy$ gives us an encoding of the construct `if-then-else`. We are now close to being able to write recursive function definitions in a syntax similar to languages like Haskell. For instance, we could aim to write

$$\begin{array}{rll} factorial\ \langle n\rangle = & \text{if } (iszero(\langle n\rangle)) & \text{then } 1 \\ & & \text{else } mult\ \langle n\rangle(factorial(pred\ \langle n\rangle)) \end{array}$$

where, of course, we still have to encode the predicate *iszero* and the arithmetic function *pred* (predecessor).

Assuming we can do this, how do we convert such a recursive definition into a lambda term?

From recursive functional definitions to lambda terms

Let $F = \lambda x_1 x_2 \ldots x_n E$ be a recursive definition of F where E contains not only the variables x_1, x_2, $\ldots$, x_n and also the expression F. How do we transform such a definition into an equivalent lambda term?

To do this, we choose a new variable f and convert E to E^* by replacing every occurrence of F in E by (ff). That is, if E is of the form $\cdots F \cdots F \cdots$ then E^* is $\cdots (ff) \cdots (ff) \cdots$. Now, write

$$\begin{aligned} G &= \lambda f x_1 x_2 \ldots x_n . E^* \\ &= \lambda f x_1 x_2 \ldots x_n . \cdots (ff)\ cdots (ff) \cdots \end{aligned}$$

Then

$$GG = \lambda x_1 x_2 \ldots x_n . \cdots (GG) \cdots (GG) \cdots$$

This means that GG satisfies the defining equation for F. We can therefore write $F = GG$, where $G = \lambda f x_1 x_2 \ldots x_n . E^*$.

Encoding minimalization

Now that we know how to convert recursive definitions into lambda terms, we can encode minimalization as follows. Assume that we want to define a function $f : \mathbb{N}^k \to \mathbb{N}$ by minimalization from a function $g : \mathbb{N}^{k+1} \to \mathbb{N}$ for which we already have an encoding $\langle g \rangle$. We begin with a recursive definition of a term F as follows.

$$\begin{aligned} F = \lambda n x_1 x_2 \ldots x_k .\ \text{if } \mathit{iszero}(\langle g \rangle\ n\ x_1\ x_2\ \ldots\ x_k)\ & \text{then } n \\ & \text{else } F(\mathit{succ}\ n)\ x_1\ x_2\ \ldots\ x_k \end{aligned}$$

Let $\tilde{F}$ be the lambda term corresponding to F after unravelling the recursive definition. The encoding $\langle f \rangle$ of f is then $\tilde{F}\ \langle 0 \rangle$.

All that remains is to provide an encoding for the predicate *iszero*.

Encoding of *iszero*

12.13 A fixed point combinator

Consider the recursive definition $F = \lambda x.x(Fx)$. By the previous "trick" for unravelling recursive definitions, we can find a lambda term for F as follows.

$$\begin{aligned} F &= GG, \text{where } G = \lambda f x . x(ffx) \\ &= \lambda f x . x(\lambda f x . x(ffx) \lambda f x . x(ffx) x) \end{aligned}$$

Notice that $FX = X(FX)$ for any lambda term X, by the definition of F. For any term Z, a fixed point of Z is a term M such that $ZM = M$. Clearly, if we set $M \equiv FZ$, we obtain a fixed point for Z. Notice that it does not matter what Z is—any lambda term Z has a fixed point FZ where F is the function we have just constructed. This fixed point operator is due to Turing and is traditionally denoted Θ.

12.14 Making sense of terms without normal forms

Recall that for terms with normal forms, the normal form denotes the "value" of the term. What about terms without normal forms? Are all terms without normal forms equally "meaningless"?

Suppose we want an equivalence $\approx$ on lambda terms such that:

1. $(\lambda x M)N \approx M\{x \leftarrow N\}$—that is, $\approx$ the equivalence induced by the β reduction.

2. If M and N do not have normal forms, then $M \equiv N$.

3. Functions that are equated by $\approx$ yield equivalent results for the same arguments. That is, if $M \approx N$ then for all R, $MR \approx NR$.

Then, we can show that $\approx$ is the trivial relation the equates all terms—in other words, $M \approx N$ for all M, N!

To see this, consider the function F defined by

$$Fxb = \text{ if } b \text{ then } x \text{ else } (Fxb)$$

As we have seen above, we can plug in our definition for `if-then-else` and then unravel this recursive definition to yield a lambda term for F.

It turns out that

$$F = GG, \text{ where } G = \lambda fxb.(\text{ if } b \text{ then } x \text{ else } (ffxb)$$

Now, consider $FX\langle\texttt{tt}\rangle$ and $FX\langle\texttt{ff}\rangle$, where $\langle\texttt{tt}\rangle$ and $\langle\texttt{ff}\rangle$ are the encodings of the truth values true and false.

- $FX\langle\texttt{tt}\rangle \rightarrow \text{if } \langle T\rangle \text{ then } X \text{ else } (FX\langle\texttt{tt}\rangle) \rightarrow X$.

- $FX\langle\texttt{ff}\rangle \rightarrow \text{if } \langle F\rangle \text{ then } X \text{ else } (FX\langle\texttt{ff}\rangle) \rightarrow FX\langle\texttt{ff}\rangle$.

Now, suppose we have an equivalence $\approx$ as described above. Then

$$\begin{array}{lll}
FZ & \rightarrow & (\lambda xb.(\text{if } b \text{ then } x \text{ else } (Fxb))Z \\
 & \rightarrow & (\lambda b.(\text{if } b \text{ then } Z \text{ else } (FZb))) \\
 & \rightarrow & (\lambda b.(\text{if } b \text{ then } Z \text{ else } G), \text{where } G = \text{if } b \text{ then } Z \text{ else } (fZB) \\
 & \rightarrow & (\lambda b.(\text{if } b \text{ then } Z \text{ else } (\text{if } b \text{ then } Z \text{ else } G)) \\
 & \rightarrow & \ldots
\end{array}$$

Since FZ does not terminate for any Z, we have $FX \approx FY$ for all X and Y. But, since $FX \approx FY$ implies that $FXM \approx FYM$ for all M (by Condition 3 in the definition of $\approx$). But then we have $FX\langle\texttt{tt}\rangle \approx FY\langle\texttt{tt}\rangle$. Since $FZ\langle\texttt{tt}\rangle \rightarrow Z$ for all Z, this means that $X \approx Y$ for all X and Y!

Chapter 13

Introducing types into the lambda calculus

So far, we have been looking at the untyped lambda calculus. What happens if we introduce types into the calculus?

13.1 Simply typed lambda calculus

In the simply typed lambda calculus, we begin with a syntax for types. Since we have no constants in our set of lambda terms, we fix a single type constant 0 that represents an atomic type. The syntax of types is then given by the grammar

$$t := 0 \mid t \to t$$

Thus, examples of types are 0, $0 \to 0$, $(0 \to 0) \to 0 \to 0$. As we have seen in Haskell, the operator $\to$ associates to the right.

We now build up typed terms inductively in such a way that only well typed terms are generated.

First, for each type s, we fix a separate set of variables. Then, we define by mutual induction the set Λ_s of lambda terms of each type s as follows:

1. For each s, every variable of type s is in Λ_s.
2. If M is a term in $\Lambda_{s \to t}$ and N is a term in Λ_s then (MN) is a term in Λ_t.
3. If M is a term in Λ_t and x is a variable of type s then $(\lambda x.M)$ is a term in $\Lambda_{s \to t}$.

Notice that every term we construct has a well defined type. Terms like (ff) that gave us some apparently contradictory behaviour earlier cannot be assigned a type (why?) and are thus not legal terms of this calculus.

The β rule for the simply typed lambda calculus is the usual one, with a check for type consistency:

$$(\lambda x.M)N \rightarrow_\beta M\{x \leftarrow N\} \quad \text{provided } x \text{ and } N \text{ are of the same type } s \text{ and } M \text{ is of type } s \rightarrow t$$

It turns out that the corresponding reduction relation $\rightarrow^*$ for this calculus is Church-Rosser. In fact, this calculus has a much stronger property—it is *strongly normalizing.*

We say that a term is *normalizing* if it has a normal form. Further, at term is *strongly normalizing* if every reduction sequence that the term admits leads to a normal form. We have seen examples of non-normalizing and non-strongly normalizing terms in the untyped calculus. For instance

- $(\lambda x.xx)(\lambda x.xx)$ is not normalizing, and
- $(\lambda yz.z)((\lambda x.xx)(\lambda x.xx))$ is not strongly normalizing.

We say that a lambda calculus is strongly normalizing if every term in the calculus is strongly normalizing. It turns out that the simply typed lambda calculus is strongly normalizing. We will not prove this result, but one intuitive explanation is that every application of β reduction strictly reduces the type complexity of the lambda term, so there is a natural limitation on the length of a reduction from a fixed starting term.

We have observed that the syntax of the simply typed lambda calculus permits only well-typed terms. What if we ask the converse question—given an arbitrary term in the untyped lambda calculus, is it well-typed? That is, is it a valid term in the simply typed lambda calculus? For instance, a term of the form (ff) cannot be assigned a sensible type and is hence clearly not definable in the simply typed calculus. It turns out that this problem is decidable.

In fact, if M is a typable term, then we can find a unique type s for M such that every possible type for M is an instance of s (that is, can be obtained by replacing some of the 0's in s by another type). Such a type is called the "principal type scheme" for M.

Observe also that the encodings $\langle m \rangle$ for natural numbers that we have provided are well typed. Hence, we can carry over much of our arithmetic from the untyped calculus to the simply typed calculus.

13.2 Polymorphic typed calculi

The simply typed lambda calculus has only explicit types. It cannot describe polymorphic functions as found, say, in Haskell. To remove this limitation, Girard and Reynolds independently invented what is called the second-order polymorphic lambda calculus (also known as Girard's *System F*).

We begin by extending the syntax of types to include type variables, that we shall denote a,b .... We assume that our lambda calculus includes constants in addition to variable, so we also permit an arbitrary set of type constants, typically denoted i, j, ...

The set of type schemes is given by the following grammar:

$$s ::= a \mid i \mid s \rightarrow s \mid \forall a.s$$

The terms of the second order polymorphic lambda calculus are then given by

1. Every variable and constant is a term.
2. If M and N are terms, so is (MN).
3. If M is a term, x is a variable and s is a type scheme, then $(\lambda x \in s.M)$ is a term.
4. If M is a term and s is a type scheme, (Ms) is a term.
5. If M is a term and a is a type variable, then $(\Lambda a.M)$ is a term.

The last two rules allow us to define functions whose types are defined in terms of type variables (like polymorphic functions in Haskell). Rule 5 says that we can make a type variable a parameter of an expression to get type abstraction, just as we make a normal variable into a parameter in lambda abstraction. Rule 4 permits us to "apply" a type abstraction to a concrete type.

In this syntax, a polymorphic identity function would be written as

$$\Lambda a.\lambda x \in a.x$$

We have two β rules, one for lambda abstraction and one for type abstraction.

- $(\lambda x \in s.M)N \rightarrow_\beta M\{x \leftarrow N\}$.
- $(\Lambda a.M)s \rightarrow_\beta M\{a \leftarrow s\}$.

It turns out that this calculus is also strongly normalizing. However, it is not known whether the type checking problem is decidable—that is, given a term, can it be assigned a sensible type. This process of assigning a type to a term without explicit type information is called *type inference.*

What is a "sensible" type? There is a natural way in which the type of a complex expression can be deduced from the types assigned to its constituent parts. One way to formalize this is to define a relation $A \vdash M : s$ where A is list $\{x_i : t_i\}$ of type "assumptions" for variables. We read this relation as follows: under the assumptions in A, the expression M has type s.

Clearly, for a variable x,

$$A \vdash x : s \text{ iff } (x : s) \text{ belongs to } A$$

We use the following notation to extend lists of assumptions. Let A be a list of assumptions. Then, $A + \{x : s\}$ is the list in which all variables other than x carry the same assumptions as in A and any assumption for x in A, if such an assumption exists, is overridden by the new assumption $x : s$.

We can now present the type inference rules for complex terms as follows:

$$\frac{A + \{x : s\} \vdash M : t}{A \vdash (\lambda x \in s.M) : s \to t}$$

$$\frac{A \vdash M : s \to t, \quad A \vdash N : s}{A \vdash (MN) : t}$$

$$\frac{A \vdash M : s}{A \vdash (\Lambda a.M) : \forall a.s}$$

$$\frac{A \vdash M : \forall a.s}{A \vdash Mt : s\{a \leftarrow t\}}$$

For instance, we can derive a type for our polymorphic identity function as follows:

$$\frac{\dfrac{x : a \vdash x : a}{\vdash (\lambda x \in a.x) : a \to a}}{\vdash (\Lambda a.\lambda x \in a.x) : \forall a.a \to a}$$

As we remarked earlier, the decidability of type inference for this calculus is open—that is, we do not have an algorithm (nor can we prove the nonexistence of an algorithm) to decide whether a given expression can be assigned a type that is consistent with the inference system described above.

How is it then that type checking is possible for languages like Haskell? The answer is that Haskell, ML and other functional programming languages use a restricted version of the polymorphic types defined here.

In Haskell, all type variable are universally quantified at the top level. Thus, if we write a Haskell type such as

```
map ::  (a -> b) -> [a] -> [b]
```

we really mean the type

$$\text{map} :: \forall a, b.\ (a \to b) \to [a] \to [b]$$

This implicit top level universal quantification of all type variables is known as *shallow typing*. In contrast, the second order polymorphic calculus, we can permits *deep typing*, with more complicated type expressions such as

$$\forall a.\ [(\forall b.\ a \to b) \to a \to a]$$

Note here that the $\forall b$ quantifier inside the expression cannot be pulled out to the top without altering the meaning of the type.

13.3 Type inference as equation solving

One way to tackle the type inference problem is to represent it as a problem of solving a system of equations involving type variables. Consider, for instance, how we intuitively assign a type to the following function definition in Haskell:

```
twice f x = f (f x)
```

We begin by observing that `twice` is a function of two arguments so, generically, we have `twice ::  a -> (b -> c)` for some unknown types `a`, `b` and `c`. We then observe that the arguments `f` and `x` occur as a function application `f x` in the right hand side, so we have the further requirement that `a`, the type of `f`, must itself be of the form `d -> e` and the type of `x` must be compatible with `d -> e`, so `b` must be the same as `d`. Then, we see that the term `(f x)` itself appears as an argument to `f` so the output type `e` of `f` must be compatible with its input type `d`. Finally, since the output of `twice` is `f (f x)`, we must have `c` compatible with `e`. We can represent this reasoning in terms of the following set of "type equations".

`a`	=	`d -> e`	(because `f` is a function)
`b`	=	`d`	(because `f` is applied to `x`
`e`	=	`d`	(because `f` is applied to `(f x)`
`c`	=	`e`	(because output of `twice` is `f (f x)`

If we "solve" these equations, we come to the conclusion that all four type variables `b`, `c`, `d` and `e` must in fact be equal to each other and `a` is of the form `b -> b` (or `c -> c` or ...). However, there is no further restriction on these variables, so we arrive at the conclusion that

```
twice :: (b -> b) -> b -> b
```

Here, the variable `b` is universally quantified. Notice that by leaving the interpretation of `b` "open", we arrived a "most general" type for `twice`.

If we have an expression involving `twice`, such as `twice f 5`, we can deduce that since 5 is of type `Int`, the type variable `b` in the type of `twice` must be uniformly assigned the value `Int`. Thus, in `twice f 5`, the (unknown) function `f` must have type `Int -> Int` and the overall type of the expression must be `Int`.

13.4 Unification

In general, for type inference we need a systematic method to perform the intuitive computation described above. The solution we seek to our system of equations is a special type of function, called a *substitution*, that maps each variable to an expression in such a way that all the equations are satisfied. In order to find the "most general" solution to a system of equations involving variables, we have to identify a substitution with "least constraints".

This problem is called *unification* and the "least constrained" substitution map that we seek is called the *most general unifier* or *mgu.* In 1965, J.A. Robinson described an algorithm that solves the unification problem. We present a version of the algorithm below.

In general, the unification problem is one of solving equations involving *terms* over a fixed *signature.* A signature consists of a collection of function symbols, each of a fixed arity. Constants can be regarded as functions with arity 0. In addition we have variables that can stand for arbitrary terms. Informally, a term is a well-formed expression consisting of function symbols and variables. Formally, terms can be defined inductively as follows:

- Every variable is a term.
- If f is a k-ary function symbol in the signature and t_1, t_2, …, t_k are terms, then $f(t_1, t_2, \ldots, t_k)$ are terms.

In the discussion that follows, we use lower case letters like a, b, f, p, …to denote function symbols and capital letters like X, Y, …to denote variables. An instance of the unification problem is a system of equations, like the following:

$$\begin{array}{rcl} f(X) & = & f(f(a)) \\ g(Y) & = & g(Z) \end{array}$$

To solve this system, we have to assign values to the variables X, Y and Z. The values we assign are terms and, as we mentioned earlier, an assignment of terms to variables is called a *substitution.* For instance, we could assign X the term $f(a)$, Y the term $g(a)$ and Z the term $g(a)$. We write such a substitution as $\{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\}$. We can give this substitution a name, for instance θ.

Given a term t and a substitution θ, the effect of applying θ to t is usually written $t\theta$ (note that this convention reverses the usual notation for function application). The term $t\theta$ is the result obtained by uniformly mapping the variables in t according to the assignments dictated by θ. For instance, in the example above, if $\theta = \{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\}$ and $t = f(X)$, then $t\theta = f(f(a))$. We can easily verify that $f(X)\theta = f(f(a))\theta$ and $g(Y)\theta = g(Z)\theta$, so θ is a solution to the unification problem given above.

Substitutions are applied in parallel, simultaneously. For instance, if we have an expression of the form $g(p(X), q(f(Y)))$ and we apply the substitution $\gamma = \{X \leftarrow Y, Y \leftarrow f(a)\}$ to this expression, we get a new expression $g(p(Y), q(f(f(a))))$. The point to note is that subexpression $p(Y)$ obtained by the assignment $X \leftarrow Y$ is not transformed transitively into $p(f(a))$ by the assignment $Y \leftarrow f(a)$ in γ.

Coming back to the example above, the substitution $\theta = \{X \leftarrow f(a), Y \leftarrow g(a), Z \leftarrow g(a)\}$ is not the only solution to the set of equations

$$\begin{array}{rcl} f(X) & = & f(f(a)) \\ g(Y) & = & g(Z) \end{array}$$

Another, simpler, solution is $\theta' = \{X \leftarrow f(a), Y \leftarrow a, Z \leftarrow a\}$. Yet another solution is $\theta'' = \{X \leftarrow f(a), Y \leftarrow Z\}$. The last solution is the "least constrained" substitution. Any other substitution that solves the problem can be broken upto into two steps, the first of which is the substitution θ''. For instance, θ is the same as θ'' followed by the substitution $\{Y \leftarrow g(a)\}$. A solution that has this property is called a *most general unifier* for the unification problem.

Not all unification problems admit a solution. For instance, if we have an equation of the form $p(Z) = q(f(Y))$ with different outermost function symbols, no substitution for Z and Y can make these terms equal.

We also have a problem with equations of the form $X = f(X)$. Since X occurs in $f(X)$, any substitution for X will also apply to the nested X within $f(X)$ and the two terms cannot be made equal.

It turns out that any set of equations in which these two problems do not occur admits a most general unifier.

A unification algorithm

The algorithm to find a most general unifier proceeds as follows.

- We begin with a set of equations $\{t_1^l = t_1^r, t_2^l = t_2^r, \ldots, t_m^l = t_m^r\}$.
- Perform the following transformations on the set of equations so long as any one of them is applicable.

  1. Transform $t = X$, where t is not a variable, to $X = t$.
  2. Erase any equation of the form $X = X$.
  3. Let $t = t'$ be an equation where neither t nor t' is a variable. Then, $t = f(\ldots)$ and $t' = f'(\ldots)$ for some function symbol f and f'.
     - If $f \neq f'$, terminate—the set of equations is not unifiable.
     - Otherwise, $t = f(t_1, t_2, \ldots, t_k)$ and $t' = f(t_1', t_2', \ldots, t_k')$, where f is of arity k. Replace the equation

       $$f(t_1, t_2, \ldots, t_k) = f(t_1', t_2', \ldots, t_k')$$

       by k new equations

       $$t_1 = t_1', t_2 = t_2', \ldots, t_k = t_k'$$
  4. Let $X = t$ be an equation such that X occurs in t. Terminate—the set of equations is not unifiable.
  5. Let $X = t$ be an equation such that X does not occur in t and X also occurs in some other equation. Transform the set of equations by replacing all occurrence of X in other equations by t.

We claim that this algorithm terminates with a most general unifier. Termination follows from the fact that the first four transformation rules can be used only a finite number of times without using rule 5. Let n be the number of distinct variables in the original set of m equations. Rule 5 can be used at most one time for each variable. Thus, overall, the five transformations can be applied only a finite number of times.

When no rules apply, every equation is of the from $X = t$ and each variable X that appears on the left hand side of an equation does not appear anywhere else in the set of equations. Thus, the resulting set of equations defines a substitution

$$\{X_1 \leftarrow t_1, X_2 \leftarrow t_2, \ldots, X_n \leftarrow t_n\}$$

We have to argue that this is a most general unifier. First, we argue that it is a unifying substitution. This follows from the fact that each of the transformations in the algorithm preserves the set of unifiers. Arguing that it is an mgu is more complicated and we omit the proof.

Here is an example of how the algorithm works. We start with the equations

$$g(Y) = X, f(X, h(X), Y) = f(g(Z), W, Z)$$

Applying rule 1 to the first equation and rule 3 to the second equation, we get

$$X = g(Y), X = g(Z), h(X) = W, Y = Z$$

Using rule 5 on the second equation, we replace X by $g(Z)$ everywhere to get

$$g(Z) = g(Y), X = g(Z), h(g(Z)) = W, Y = Z$$

Applying rule 3 to the first equation we get

$$Z = Y, X = g(Z), h(g(Z)) = W, Y = Z$$

Using rule 5 on the last equation to replace Y by Z and eliminating the resulting equation $Z = Z$ by rule 2, we get

$$X = g(Z), h(g(Z)) = W, Y = Z$$

Finally, we reverse the second equation by rule 1 to get

$$X = g(Z), W = h(g(Z)), Y = Z$$

No further rules apply, so the most general unifier is $\{X \leftarrow g(Z), W \leftarrow h(g(Z)), Y \leftarrow Z\}$.

The algorithm we have described is different from Robinson's original algorithm, whose correctness is more difficult to establish. The most efficient algorithms for unification work in linear time.

13.5 Type inference with shallow types

Consider the basic lambda calculus, enhanced with constants. We have some built-in types $i, j, k, \ldots$ and a set of constants C_i for each built-in type i. For instance, if i is the type `Char`, C_i is the set of character constants. The syntax of lambda terms is then

$$\Lambda = c \mid x \mid \lambda x.M \mid MN$$

where $c \in C_i$ for some built-in type i, x is a variable and $M, N \in \Lambda$.

To infer types for these terms, we can set up equations involving type variables inductively as follows. Let $M \in \Lambda$ be a lambda term. Then:

- If M is a constant $c \in C_i$, the type of M is i.
- If M is a variable x, assign a fresh type variable α to x.
- If M is of the form $\lambda x.M'$, assign M the type $\alpha \to \beta$ for fresh type variables α and β. Inductively, let γ be the type of x in M'. Then, the input type α of $\lambda x.M'$ should match the type of x, so add the equation $\alpha = \gamma$ to the set of type equations.
- If M is of the form $M'N'$, we must inductively have assigned M' a type of the form $\alpha \to \beta$ and N' a type of the form γ. Assign M the type β and add the equation $\alpha = \gamma$ to the set of type equations to enforce that this application is well-typed.

With this syntax, suppose we want to write a function equivalent to the following Haskell definition.

```
applypair f x y = (f x,f y)
```

In this definition, we might like to permit f to be a polymorphic function and `x` and `y` to be of different types. For instance, we might ask for the following expression to be well typed, where `id` is the identity function `id z = z`.

```
applypair id 7 'c' = (id 7, id 'c') = (7,'c')
```

However, if we try to assign a type to this function, we find that we have the following set of constraints, which cannot be unified.

```
id  :: a -> a
7   :: Int
'c' :: Char
a = Int           (from id 7)
a = Char          (from id 'c')
```

In fact, the type that Haskell assigns to `applypair` is `(a -> b) -> b -> b -> (b,b)`. To see why this is so, let us look how this function would be defined in the version of the lambda calculus we have just defined. The corresponding expression is

$$\lambda fxy.\mathit{pair}\ (fx)(fy) \text{ where } \mathit{pair} \equiv \lambda xyz.(zxy)$$

As the type inference rules we have provided suggest, when we pass a function to this term for the argument f with type $\alpha \to \beta$, the value of α will have to be unified with the types of both x and y because of the expressions (fx) and (fy) in the body of the expression. This then forces x and y to have the same type.

One way to permit the definition of the richer version of `applypair` is to extend the syntax of lambda terms to allow local definitions as follows.

$$\Lambda = C_i \mid x \mid \lambda x.M \mid MN \mid \mathsf{let}\ f = e\ \mathsf{in}\ M$$

Here $f = e$ in *let ... in ...* provides a local declaration. We can now define the lambda term for `applypair id` as

$$\mathsf{let}\ f = \lambda z.z\ \mathsf{in}\ \lambda xy.\mathit{pair}\ (fx)(fy)$$

which translates into Haskell as

```
  applypair x y = (f x,f y) where f z = z
```

or, equivalently,

```
  applypair x y = let f z = z in (f x,f y)
```

What is the type inference rule for *let ... in ...*? Here is a first attempt at formulating the rule.

- Let $M = \mathsf{let}\ f = e\ \mathsf{in}\ M'$ where f inductively has type t. Let $\{\alpha, \beta, \ldots\}$ be the set of type variables that occur in t. Then for each instance of f in M', make *copies* of these variables. Thus, the first instance of f in M' will be assigned type t with the variables modified to $\alpha_1, \beta_1, \ldots$, the second instance will be assigned type t with the variables modified to $\alpha_2, \beta_2, \ldots$, and so on.

With this rule, we find that the two occurrences of f in $\mathsf{let}\ f = \lambda z.z\ \mathsf{in}\ \lambda xy.\mathit{pair}\ (fx)(fy)$ have types $\alpha_1 \to \beta_1$ and $\alpha_2 \to \beta_2$. The expression fx unifies the type variable α_1 with the type of x while the expression fy unifies the type variable α_2 with the type of y. However, since α_1 and α_2 are *different* variables, this does not force the type of x to match the type of y.

Notice that from the point of view of β-reduction, the terms $\mathsf{let}\ f = e\ \mathsf{in}\ \lambda x.M$ and $(\lambda fx.M)e$ are equivalent. But, as we have seen, the type inference rules for the two expressions are quite different.

In a more extreme example, one form may be typable while the other is not. Consider the expressions $(\lambda I.(II))(\lambda x.x)$ and $\textsf{let } I = \lambda x.x \textsf{ in } (II)$. Here, the first form cannot be typed because no self-application ff can be typed. However, in the *let ... in ...* version, the type of $\lambda x.x$ is $\alpha \rightarrow \alpha$ and this is copied as $\alpha_1 \rightarrow \alpha_1$ and $\alpha_2 \rightarrow \alpha_2$ for the copies of I in the body of the function. We can then unify α_1 with $\alpha_2 \rightarrow \alpha_2$ to derive that the first I in II has the type $(\alpha_2 \rightarrow \alpha_2) \rightarrow (\alpha_2 \rightarrow \alpha_2)$ and, overall, II has type $(\alpha_2 \rightarrow \alpha_2)$.

There is a subtlety that we have overlooked in our type inference rule for *let ... in ...*. Consider the function

```
applypair2 w x y = ((tag x),(tag y))
  where
    tag      = pair w
    pair s t = (s,t)
```

Intuitively, it is clear that the type of `applypair2` is `a -> b -> c -> ((a,b),(a,c))` because `applypair2` constructs the pair of pairs `((w,x),(w,y))` from its three inputs `w`, `x` and `y`.

However, if we apply our type inference strategy, we begin with the types:

```
applypair2 :: a -> b -> c -> (d,e)
pair       :: f -> g -> (f,g)
tag        :: h -> (i,h)
```

Using our variable copying rule for *let ... in ...*, we get that the two instances of `tag` in `applypair2` have type `d = h1 -> (i1,h1)` and `e = h2 -> (i2,h2)`. The application `tag x` gives us the constraint `h1 = b` and the application `tag y` gives us the constraint `h2 = c`. The application `pair w` gives us the constraint `a = i`, but this constraint *is not propagated* to the copies `i1` and `i2` that we have made of `i` when expanding the type of `applypair2`. Thus, we get the type `applypair2 :: a -> b -> c -> ((i1,b),(i2,c))` instead of the correct type `a -> b -> c -> ((a,b),(a,c))`.

Clearly, the problem lies in the copies that we made of the type variable `i` when instantiating the types for the two copies of `tag`. The distinction between the variables `h` and `i` that appear in the type of `tag` is that the variable `i` is unified with the type of one of the arguments to the main function within which `tag` is defined. In the literature, variables like `h` that are "free" within the local definition are called *generic* variables. The corrected type inference rule for *let ... in ...* requires that copies be made only for generic variables.

- Let $M = \textsf{let } f = e \textsf{ in } M'$ where f inductively has type t. Let $\{\alpha, \beta, \ldots\}$ be the set of *generic* type variables that occur in t. Then for each instance of f in M', make *copies* of these generic variables. Thus, the first instance of f in M' will be assigned type t with the generic variables modified to $\alpha_1, \beta_1, \ldots$, the second instance will be assigned type t with the generic variables modified to $\alpha_2, \beta_2, \ldots$, and so on. The non-generic type variables in t retain their identity in all instances of f.

If we apply this new rule when assigning a type to `applypair2`, we observe that the type variable `i` should not be duplicated. This results in the type `a -> b -> c -> ((i,h1),(i,h2))` being assigned to `applypair2` after copying the generic variables of the type of `tag` into the two instances of `tag` that occur in the body of `applypair2`. The constraints `a = i`, `b = h1` and `c = h2` then result in the correct type `a -> b -> c -> ((a,b),(a,c))` being assigned to `applypair2`.

The type inference strategy that we have sketched here can be efficiently implemented using a set of inference rules in the same style as those that we described for the second-order polymorphic typed lambda calculus in an earlier section. This algorithm was first described by Milner (he called it Algorithm W) and was implemented in the programming language ML and, subsequently, in other typed functional languages like Haskell. The important fact is that for these languages, type inference is decidable, so the programmer can write function definitions without providing explicit type information. The type inference system can then recognize whether the expression is well-typed. Freeing the programmer from the burden of providing explicit types makes these strongly typed functional languages significantly easier and more attractive to use. In a sense, the combination of *let ... in ...* definitions in the lambda calculus and shallow types seems to provide an optimum combination of expressiveness and useability in the typed lambda calculus.

Part VII

Logic programming

Chapter 14

Introduction to logic programming

Logic programming is a probably more accurately called "programming with relations", in contrast to functional programming, which is "programming with functions".

The idea is that a function f with n arguments can always be written as a relation R_f with n+1 arguments:

$$f(x_1, x_2, \ldots, x_n) = y \text{ iff } (x_1, x_2, \ldots, x_n, y) \in R_f$$

Having gone from functions to relations, we can generalize our objectives and give rules to compute not only those special relations that correspond to functions (where the last component of each tuple is uniquely fixed by the others), but more general relations as well.

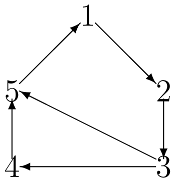
We will describe logic programming using a syntax that is based on Prolog, the most commonly used programming language in this paradigm.

14.1 Facts and rules

We start with two types of simple entities, variables and constants. The convention is that variables start with a capital letter—`X`, `Y`, `Name`, ..., while constants start with a lower case letter—`ball`, `node`, `graph`, `a`, `b`, ...

Constants are, in general, "uninterpreted"—that is, they are not further organized into types like `Char` and `Float` with underlying meaning. However, for convenience, we will allow ourselves to use natural numbers as a special type of constant and compute arithmetic expressions over these numbers.

A Prolog program defines a relation through facts and rules. For instance, consider a directed graph with five nodes.



We can represent its edge relation using the following *facts*.

```
edge(3,4).
edge(4,5).
edge(5,1).
edge(1,2).
edge(3,5).
edge(3,2).
```

Each fact just lists out explicitly a concrete tuple that belongs to the relation `edge`.

We can now define another relation `path` using a set of *rules*, as follows:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

These rules are to be read as follows:

Rule 1 For all `X`,`Y`, `(X,Y)` belongs to `path` if `(X,Y)` belongs to `edge`.

Rule 2 For all `X`,`Y`, `(X,Y)` belongs to `path` if there exists `Z` such that `(X,Z)` belongs to `edge` and `(Z,Y)` belongs to `path`.

Thus, each rule is of the form "Conclusion if Premise$_1$ and Premise$_2$ ... and Premise$_n$". This special type of logical formula is called a Horn Clause. In Prolog, the "if" is written as `:-` and the ands on the right hand side are written as commas. The left hand side of a rule is called the goal.

Notice the implicit quantification of variables—variables that appear in the goal are universally quantified (for example, `X` and `Y` above) while variables that appear in the premises are existentially quantified (for example, `Z` above).

14.2 Computing in Prolog

To start off a computation in Prolog, we ask a query. For instance, we may ask, at the Prolog interpreter prompt `?-` whether there is a path from 3 to 1, as follows

```
?- path(3,1).
```

To answer this, Prolog will try to use the rules given above and see if it can find a satisfying assignment for all the variables that answers this query. In this example, it scans the rules for `path(X,Y)` from top to bottom and proceeds as follows:

- `(3,1)` is not in `edge`, so skip Rule 1.
- Rule 2 generates two subgoals. Does there exists `Z` such that

  - (3,Z) belongs to edge and
  - (Z,1) belongs to path.

  Prolog tries to satisfy the first subgoal by scanning the edge relation from top to bottom and finds edge(3,4), so Z is set to 4.

  The second goal then becomes path(4,1). Since (4,1) is not in edge, this again generates two subgoals: Does there exist Z' (a new variable, not the old Z) such that

  - (4,Z') belongs to edge and
  - (Z',1) belongs to path.

  Once again, to satisfy the first goal we scan edge from the top and find edge(4,5), so Z' is set to 5.

  The second goal then becomes path(5,1). Now, (5,1) does belong to edge, so path(5,1) is true.

  We can now backtrack and report success—both the subgoals for path(4,1) have succeeded, path(4,1) is true.

  Backtracking once more, both the subgoals for the original goal have succeeded, so path(3,1) is true.

Essentially, Prolog does a depth first search through all possible satisfying assignments to the variables in the premises in order to satisfy the goal. In this case, it found a path of length three, even though there is a shorter path of length two. This is because of the way that the facts in edge are arranged. If we had placed the fact edge(3,5) before edge(3,4), Z would have been set to 5 in the first search itself. (What would happen if we place edge(3,2) before both edge(3,4) and edge(3,5)?)

14.3 Complex structures in Prolog

In general, we can represent arbitrary structures in Prolog by using nested terms. For instance, we can represent a Pascal record or a C struct of the form:

```
personal_data{
   name : amit
   date_of_birth{
      year  : 1980
      month : 5
      day   : 30
   }
}
```

by a Prolog term

```
personal_data(name(amit),date_of_birth(year(1980),month(5),day(30)))
```

The list type is built in to Prolog. Analogous to list decomposition by the expression `(head:tail)` in Haskell, we can decompose a list in Prolog as `[Head|Tail]`. For instance, we can write a function to check if an item belongs to a list as follows:

```
member(X,[X|T]).
member(X,[Y|T]) :- member(X,T).
```

In this function, for the first rule to succeed, X must match the head of the list. Formally, this is done using unification, which we have seen in our discussion of type inference. The important thing to note is that unification permits implicit pattern matching of more than one part of the goal, a feature that we noted was not permitted in patterns as used in Haskell definitions. The second rule can be invoked only if the first rule fails, in which case we are guaranteed that the head of the list does not match X, so X and Y are different. In this case, we make a recursive call.

Notice that the names T in the first rule and Y in the second rule are not really used. Prolog allows the use of a dummy variable _ in such cases.

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

All other variables are uniformly substituted in a rule—if X appears in many places, all X's will be replaced by the same value. However, for convenience, the "don't care" variable _ is not subject to this limitation, so each occurrence of _ in a rule can take on a different value. Thus, we can use the same don't care variable _ in many places in the same rule without implicitly asserting that all these positions acquire the same value. This is similar to the use of _ in Haskell patterns.

14.4 Turning the question around

Since Prolog deals with relations and not functions, we can invoke inverse computations in a natural way, unlike functional programming. For instance, in our first example, we can ask

```
?- edge(3,X).
```

This asks for all values of X that satisfy the relation edge. If we respond with ; to each answer provided, Prolog will exhaustively list out all possible values that X can take, in the order in which they are defined. The response would look something like

```
    X=4;
    X=5;
    X=2;
    No.
```

The final `No.` indicates that there are no further ways of satisfying the goal. Likewise

```
?- path(3,Y).
```

will exhaustively list out all values for Y that make this relational assertion true.

14.5 Arithmetic

A first attempt to write a program to compute the length of a list would look as follows.

```
length([],0).
length([H,T],N) :- length(T,M), N = M+1.
```

If we run this program, we get a somewhat surprising result.

```
?- length([1,2,3,4],N).
N=0+1+1+1+1
```

This is because `=` signifies unification, not equality. Thus, when the base case is reached—that is, `length(T,M)` with `T` equal to the empty list—`M` is assigned `0`. The phrase `N=M+1` unifies `N` with `M+1` and sets `N` to the expression `0+1`. As the program pops out of each level of recursion, an extra `+1` is added, resulting in the final expression that we saw above.

To match arithmetic expressions based on their value, Prolog provides the connective `is`. Thus, `length` should be defined as follows.

```
length([],0).
length([H,T],N) :- length(T,M), N is M+1.
```

Now, the query `length([1,2,3,4],N)` behaves as we would expect.

```
?- length([1,2,3,4],N).
N=4
```

Here is a more convoluted version of `length`.

```
length(L,N) :- auxlength(L,0,N).
auxlength([],N,N).
auxlength({H,T],M,N) :- auxlength(T,M1,N), M1 is M+1.
```

Consider what happens when we ask for

```
?- length([0,1,2],N)
```

The goals generated are

```
auxlength([0,1,2],0,N) -> auxlength([1,2],1,N) ->
auxlength([2],2,N) -> auxlength([],3,N) -> auxlength([],3,3)
```

so Prolog reports back that

```
N = 3.
```

Observe the role played by unification—the variable N is being propagated through all the subgoals and the final substitution works its way back to the original query.

14.6 Negation as failure

In Prolog, we can write a premise of the form not(R(X,Y,Z)). This becomes true if Prolog is not able to justify R(X,Y,Z). In other words, for each relation Prolog uses a "closed world" assumption—the rules and facts exhaustively determine all the members of the relation and anything that is not computable as being part of the relation is out of the relation. This approach is called "negation as failure".

Negation has some unexpected effects. For instance, consider the following query.

```
?- not(X=5).
No.
```

The No is justified as follows. To check not(X=5), Prolog first tries the goal X=5. Since X can be unified with 5, this goal succeeds. Therefore, not(X=5) fails.

Now, consider a query to extract from a list Ls all elements that are not equal to 1. We could write it two ways:

```
?-  not(X=1), member(X,Ls).

?-  member(X,Ls), not(X=1).
```

In the first version, not(X=1) always fails, as we saw above, because X is not bound to any value initially. This results in the query failing. The second query, however, does the job that we want because X is first bound to an element of Ls before being tested for X=1.

14.7 Cut

Prolog provides a somewhat controversial mechanism to control backtracking. If we insert an exclamation mark `!` in the set of premises, backtracking will not cross the exclamation mark.

For instance, if we rewrote path above as

```
path(X,Y) :- edge(X,Z),!, path(Z,Y).
```

then for each `X`, it would try `path(Z,Y)` only once, for the first edge of the form `edge(X,Z)`. If there is no path using this value of `Z` it will not backtrack to another edge involving `X`—the overall search just fails.

Cut is useful for optimizing Prolog code to avoid unnecessary backtracking. Suppose we want to program the ternary predicate `ifthenelse(B,X,Y)` that evaluates `X` if `B` is true (that is, if the goal `B` succeeds) and evaluates `Y` otherwise. A first attempt would be to write.

```
ifthenelse(B,X,Y) :- B,X.
ifthenelse(B,X,Y) :- not(B),Y.
```

We have to look at Prolog backtracking a little more to explain this code. If the first rule fails for a goal, Prolog will try alternative rules before backtracking along the first rule. Thus, if we omit the `not(B)` in the second rule, we have a problem. If `B` succeeds and `X` fails, Prolog backtracking would directly try to satisfy `Y`. By placing a guard `not(B)` on the second clause, we ensure that this does not happen. However, evaluating this guard involves an unnecessary computation because, if we reach this stage, we have already evaluated `B` so we already "know" the result `not(B)`.

A second source of inefficiency in this definition lies in the first rule. Suppose that `B` succeeds and `X` fails. We should then conclude that the `ifthenelse(B,X,Y)` fails. However, if there are multiple ways of satisfying `B`, Prolog backtracking will retry `X` for all of these options before abandoning the original goal!

We can fix both these inefficiencies by introducing a cut in the first rule.

```
ifthenelse(B,X,Y) :- B,!,X.
ifthenelse(B,X,Y) :- Y.
```

The cut has two effects. First, as we mentioned earlier, it prevents backtracking to `B`. Second, it discards all future alternative rules for this goal. Thus, even if `X` fails, the cut ensures that `Y` is not tried. On the other hand, if `B` fails, which is before the cut, normal backtracking takes Prolog to the second rule and tries `Y` without wastefully computing `not(B)` as in the previous version.

In more complex context, it is quite difficult to predict the portion of the search tree that is pruned by such a cut, so this feature is to be used with caution.

Part VIII

Scripting languages

Chapter 15

Programming in Perl

For the most part, we have focussed on features in programming languages that add structure and discipline to the process of programming. The fundamental idea has always been to achieve suitable degree of abstraction. To ensure the integrity of the abstraction, we have seen different approaches to defining a strong notion of type combined with the flexibility to define polymorphic functions.

At the other end of the spectrum we have scripting languages that impose minimal programming discipline but are ideal for rapid deployment of programs that attack "routine" tasks. One of the most successful scripting languages is Perl, an imperative language whose control flow is along the lines of C, but with several interesting features that make it ideal for text processing.

15.1 Scalar datatypes

Values associated with scalar variables in Perl are either numbers or character strings. Numbers correspond to double precision floating point numbers. Variable names begin with the special character `$`. For instance, we could have the following assignments in a Perl program.

```
$num1 = 7.3;
$str1 = "Hello123";
$str2 = "456World";
```

Variables (and their types) do not have to be declared. Moreover, the type of a variable changes dynamically, according to the type of expression that is used to assign it a value. We use normal arithmetic operators to combine numeric values. The symbol `.` denotes string concatenation. If a numeric value is used in a string expression, it is converted automatically to the string representation of the number. Conversely, if a string is used in a numeric expression, it is converted to a number based on the contents of the string. If the string begins with a number, the resulting value is that number. Otherwise, the value is 0. Continuing our example above, we have the following (text after `#` is treated as a comment in Perl).

```
$num3 = $num1 + $str1;          # $num3 is now 7.3 + 0 = 7.3
$num4 = $num1 + $str2;          # $num4 is now 7.3 + 456 = 463.3
$str2 = $num4 . $str1;          # $str2 is now "463.3Hello123"
$str1 = $num3 + $str2;          # $str1 is now the number 470.6
$num1 = $num3 . "*" . $num4;    # $num1 is now the string "7.3*463.3"
```

By default, Perl values are made available to the program the moment they are introduced and have global scope and are hence visible throughout the program. Uninitialized variables evaluate to 0 or "" depending on whether they are used as numbers or strings. The scope of a variable can be restricted to the block in which it is defined by using the word my when it is first used.

```
my $a = "foo";
if ($some_condition) {
    my $b = "bar";
    print $a;           # prints "foo"
    print $b;           # prints "bar"
}
print $a;               # prints "foo"
print $b;               # prints nothing; $b has fallen out of scope
```

15.2 Arrays

Perl has an array type, which is more like a list in Haskell because it can grow and shrink dynamically. However, unlike arrays in Java or C and lists in Haskell, Perl arrays can contain a mixture of numeric and string values. Collectively, a Perl array variable starts with the symbol @, thus distinguishing it from a scalar. An array can be explicitly written using the notation (element1,element2,...,elementk).

Perl supports array like notation to access individual elements of an array—array indices run from 0, as in C or Java. The element at position $i in array @list is referred to as $list[$i], *not* @list[$i]. A useful mnemonic is that an element of an array is a scalar, so we use $, not @.

The [...] notation can be used after the @ name of the array to denote a sublist. In this form, we provide a list of the indices to be picked out as a sublist. This list of indices can be in any order and can have repetitions.

```
@list = ("zero",1,"two",3,4);   # Note the mixed types
$val1 = $list[3];               # $val1 is now 3
$list[4] = "four";              # @list is now ("zero",1,"two",3,"four")
@list2 = @list[4];              # @list2 is ("four")
@list3 = @list[4,1,3,4,0];      # @list3 is ("four",1,3,"four",0)
```

A key fact about Perl lists is that they are always flat—nested lists are automatically flattened out. Thus, we don't need to write functions such as append to combine lists.

```
@list1 = (1,"two");
@list2 = (3,4);
@list  = (@list1,@list2);      # @list is (1,"two",3,4)
$list  = ("zero",@list);       # @list is now ("zero",1,"two",3,4)
```

The example above shows that we can use a list variable on the left hand side of an assignment. We can also combine scalar variables into a list to achieve a multiple parallel assignment. If we use a list variable as part of the multiple assignment on the left hand side it "soaks" up all the remaining values.

```
($first,$second) = ($list[0],$list[1]);
($first,@rest)   = @list;           # $first is $list[0]
                                    # @rest is @list[1,2,...]
($first,@rest,$last) = @list;       # $first is $list[0]
                                    # @rest is @list[1,2,...]
                                    # $last gets no value
($this,$that)    = ($that,$this);   # Swap two values!
```

Perl makes a distinction between *list context* and *scalar context*. The same expression often yields different values depending on the context in which it is placed. One example of this is that a list variable `@list`, when used in scalar context, evaluates to the length of the list.

```
@n = @list;                   # Copies @list into @n
$n = @list;                   # $n is the length of @list
$m = $list[@list-1];          # $m is the last element of @list
```

Perl has several builtin functions to manipulate arrays. For instance, `reverse @list` reverses a list. The function `shift @list` removes and returns the leftmost element of `@list`. The function `pop @list` removes and returns the last element of `@list`. The function `push(@list,$value)` adds `$value` at the end of `@list` while `unshift(@list,$value)` adds `$value` at the beginning of `@list`.

15.3 Control flow

Perl supports the normal branching and looping constructs found in C or Java:

- `if (condition) {statement-block} else {statement-block}`
- `while(condition) {statement-block}`
- `for (init; condition; step) {statement-block}`

The only thing to remember is that in Perl the braces around the statement blocks are compulsory, even they contain only one statement.

When evaluating the boolean condition in these statements, the number `0`, the strings `"0"` and `""` and the empty list `()` are all treated as false. All other values are true.

Because Perl does not have explicit declarations of numeric and string variables, there are separate comparison operators for numeric and string values. The symbolic operators `<`, `<=`, `>`, `>=`, `==` and `!=` compare their arguments as numeric values while string comparison is denoted by the operators `lt`, `le`, `gt`, `ge`, `eq` and `ne`.

In addition to the basic compound statements above, Perl supports an abbreviation `elsif` for mulitply branching `if` statements, a version of `if` called `unless` and a version of `while` called `until`, both of which invert the sense of the condition (that is, the statement block is executed if the condition is false).

- `if (condition) {statement-block} elsif {statement-block}`
  `elsif {statement-block} ...  else {statement-block}`
- `unless (condition) {statement-block}`
- `until (condition) {statement-block}`

For stepping through a list, Perl provides a special loop called `foreach`.

```
foreach $value (@list){        # $value is assigned to each item in
   print $value, "\n";         # @list in turn
}
```

15.4 Input/Output

Files can be opened for input or output using the `open()` function. As in languages like C, the `open()` statement associates a filehandle with a filename and a *mode* for the file, namely input, output or append.

Here are some examples. In these examples the function `die` exits after printing a diagnostic. It is separated by an `or` whose semantics is that the second part of the `or` is executed only if the first part fails.

```
open(INFILE,  "input.txt")   or die "Can't open input.txt";
  # Open in read mode -- could also write open(INFILE,"<input.txt")
open(OUTFILE, ">output.txt") or die "Can't open output.txt";
  # > indicates open in (over)write mode
open(LOGFILE, ">>my.log")    or die "Can't open logfile";
  # >> indicates open in append mode
```

Reading a file is achieved using the `<>` operator on a filehandle. In scalar context it reads a single line from the filehandle, while in list context it reads the whole file in, assigning each line to an element of the list:

```
$line  = <INFILE>;  # Read one line from INFILE into $line
@lines = <INFILE>;  # Read all lines from INFILE into list @lines
```

The `<>` operator is most often seen in a `while` loop:

```
while ($line = <INFILE>) {     # assigns each line in turn to $line
  print "Just read in this line:", $line;
}
```

The example above also introduces the `print` statement. The simplest form of the `print` statement takes a list of string arguments separated by commas. `print` can also take an optional first argument specifying which filehandle to print to:

```
print "Hello, world";
print "Hello ", $name,"\n";
print OUTFILE $record;
print LOGFILE $logmessage;
```

To close a file, we call the function `close` with the corresponding filehandle.

15.5 Matching and regular expressions

We mentioned in the beginning that Perl is well suited to text manipulation. A particularly useful feature in this regard is a powerful matching operator, that extends naturally to a search-and-replace operator.

The match operator is `=~` which allows us to match a string variable against a pattern, usually delimited by `/`. For instance, to print all lines from a file that contain the string `CMI`, we would write:

```
while ($line = <INFILE>) {
  if ($line =~ /CMI/){
     print $line;
  }
}
```

More generally, the pattern could be a regular expression. The syntax for regular expressions is similar to that in text editors like `vi` or `emacs` or in the `grep` command of Unix. In a regular expression, a character stands for itself. A sequence of characters in square brackets stands for a choice of characters. Thus, the pattern `/[Cc][Mm][Ii]/` would match any combination of lower and upper case letters that make up the string `cmi`—for instance, `CMI`, `CmI`, ... The character `.` is special and matches *any* character, so, for instance, the pattern `/[Cc].i/` would match any three letter pattern beginning with `C` or `c` and ending with `i`. We can specify a case-insensitive search by appending the modifier `i` at the end of the pattern.

```
if ($line =~ /CMI/i){       # Same as ($line =~ /[Cc][Mm][Ii]/)
```

Perl provides some special abbreviations for commonly used choices of alternatives. The expression `\w` (for *word*) represents any of the characters `_`,`a`,...,`z`,`A`,...,`Z`,`0`,...,`9`. The expression `\d` (for *digit*) represents `0`,...,`9`, while `\s` represents a whitespace character (space, tab or newline).

Repetition is described using `*` (zero or more repetitions), `+` (one or more repetitions) and `?` (zero or one repetitions). For instance the expression `\d+` matches a nonempty sequence of digits, while `\s*a\s*` matches a single `a` along with all its surrounding white space, if any. More controlled repetition is given by the syntax `{m,n}`, which specifies between `m` and `n` repetitions. Thus `\d{6,8}` matches a sequence of 6 to 8 digits.

A close relative of the match operator is the search and replace operator, which is given by `=~ s/pattern/replacement/`. For instance, we can replace each tab (`\t`) in `$line` by a single space by writing

```
 $line =~ s/\t/ /;
```

More precisely, this replaces the *first* tab in `$line` by a space. To replace *all* tabs we have to add the modifier `g` at the end, as follows.

```
 $line =~ s/\t/ /g;
```

Often, we need to reuse the portion that was matched in the search pattern in the replacement string. Suppose that we have a file with lines of the form

```
phone-number name
```

which we wish to read and print out in the form

```
name phone-number
```

If we match each line against the pattern `/\d+\s*\w.*/`, then the portion `\d+` would match the phone number, the portion `\s*` would match all spaces between the phone number and the first part of the name (which could have many parts) and the portion `\w.*` would match the rest of the line, containing all parts of the name. We are interested in reproducing the phone number and the name, corresponding to the first and third groups of the pattern, in the output. To do this, we group the portions that we want to "capture" within parentheses and then use `\1`, `\2`, ... to recover each of the captured portions. In particular, if `$line` contains a line of the form `phone-number name`, to modify it to the new form `name phone-number` we could write

```
$line =~ s/(\d+)\s*(\w.*)/\2 \1/;  # \1 is what \d+ matches,
                                   # \2 is what \w.* matches
```

One thing to remember is that if we assigned a value to `$line` using the `<>` operator, then it would initially have a trailing newline character. In the search and replace that we wrote above, this newline character would get included in the pattern `\2`, so the output would have a new line between the name and the phone number. The function `chomp $line` removes the trailing newline from `$line`, if it exists, and should always be used to strip off unwanted newlines when reading data from a file.

15.6 Associative arrays

In addition to regular arrays indexed by position, Perl has another collective type called an associative array, or a *hash*. A hash is a set of pairs of the form *(key,value)*. The keys of a hash are strings. Collectively, the name of a hash starts with a %, unlike $ for scalars and @ for arrays. To access an element of a hash we use braces rather than square brackets. The function keys extracts the keys from a hash table. Here is an example.

```
$phone{"CMI"} = 28157814;
$phone{"IMSc"} = 22541856;
$phone{"Home"} = 24404396;
foreach $k (keys %phone){
  print $k, ":", $phone{$k}, "\n";
}
```

The output generated by this will be something like

```
CMI : 28157814
Home : 24404396
IMSc : 22541856
```

Observe that the sequence in which keys listed out the keys of %phone is not the same as the sequence in which the keys were added to the hash. This usually happens in a Perl hash, due to internal optimization.

Here is a typical use of a hash. We read data about students' marks in a course from a file in which each line is of the form rollnumber marks. The file records marks from multiple evaluations (tests, assignments, ...) so each rollnumber appears more than once, in general. We split each line of input using the builtin function split that splits a string along whitespace boundaries and returns a list whose elements are the constituents of the string after splitting. We accumulate the marks for each student in a hash and print out a summary at the end.

```
while ($line = <MARKS>){
  ($rollno, $marks) = split $line;  # split $line on whitespace boundary
  $eval{$rollno} += $marks;         # Add $marks to $eval{$rollno}
}

foreach $k (keys %eval){
  print $k, ":", $eval{$k}, "\n";
}
```

The function split can also take another argument, which is a pattern describing the delimiter between fields. For instance, the following line splits a line in which values are separated by commas, where we want to also ignore the spaces before and after commas.

```
@values = split /\s*,\s*/,$line;
```

15.7 Writing functions

A function in Perl begins with the keyword `sub` followed by a name. The interesting feature of functions in Perl is that there is no explicit declaration of the list of arguments to the function. When a function is invoked, the list of arguments, if any, is made available to the function in a specially named list, `@_`. The function can then `shift` through `@_` to extract all the values that are passed to it.

For instance, here is a function that returns the maximum of exactly two values:

```
sub max {
 my ($a,$b,@rest) = @_;     # Get first arguments from @_ in $a, $b
 if ($a > $b) {return $a};
 return $b;
}
```

This function does not check whether `@_` does in fact have two valid values. No matching is done while invoking a function, since there is no *template* associated with a function. Thus, the function `max` defined above could be invoked, legally, as `max()`, or `max(1)` or `max($x,$y)` or `max(@list)` or . . . . Here is a more generic version of max, that works "properly" provided it has at least one argument. Recall that for an array `@list`, the value `@list` in scalar context gives the length of the array.

```
sub max {
 my ($tmp);                   # Declare $tmp as a local variable

 unless (@_) return 0;        # Return 0 if input list is empty

 my $max = shift @_;          # Set $max to first value in @_
 while (@_){                  # Compare $max to each element in @_
   $tmp = shift @_;
   if ($tmp > $max){$max = $tmp;}
 }
 return $max;
}
```

15.8 Sorting

An extremely useful builtin function in Perl is `sort`, which sorts an array. The default behaviour of `sort` is to use alphabetic sort. So, for instance, to read a list of phone numbers where each line is of the form `name:phone number` and print them out in alphabetical order we could write:

```
while ($line = <PHONE>){
  ($name,$phone) = split /:/,$line;
  $phonehash{$name} = $phone;       # Store phone number in a hash
}

foreach $k (sort keys %phonehash){
   print $k, ":", $phonehash{$k},"\n");
}
```

Here, we sort the list generated by `keys %phonehash` before printout out the values in the hash.

What if we want to supply a different basis for comparing values? We can supply `sort` with the name of a comparison function. Then comparison function we supply will be invoked by `sort`. Instead of using `@_` to pass parameters to the comparison function (as with normal functions), `sort` will pass the values as `$a` and `$b`. The comparison function should return −1 if the first argument, `$a`, is smaller than the second argument, `$b$`, 0 if the two arguments are equal, and 1 if the second argument is smaller than the first. So, we can sort using a numeric comparison function as follows.

```
foreach $k (sort bynumber keys %somehash){
  ...
}

sub bynumber {
  if ($a < $b) {return -1};
  if ($a > $b) {return 1};
  return 0;
}
```

In fact, this is so common that Perl supplies a builtin "spaceship" operator `<=>` that has this effect.

```
sub bynumber {
  return $a <=> $b;  # Return -1 if $a < $b,
                     #        +1 if $a > $b,
                     #         0 if $a == $b
}
```

The operator `cmp` achieves the same effect as `<=>` but using string comparison instead of numeric comparison.

To sort in descending order, we simply invert the position of the arguments in the comparison function.

```
foreach $k (sort bynumdesc keys %somehash){
  ...
}

sub bynumberdesc {return $b <=> $a;}
```

We can also use the arguments `$a` and `$b` in more complex ways. For instance, to print out a hash based on the numeric sorted order of its values, we can write:

```
foreach $k (sort bystrvalue keys %somehash){
  ...
}

sub bystrvalue {return $somehash{$a} cmp $somehash{$b};}
```

Finally, we can avoid using a separate named comparison function by just supplying the expression that is to be evaluated in braces, as follows:

```
foreach $k (sort {$a <=> $b} keys %somehash){  # Same as bynumber
foreach $k (sort {$b <=> $a} keys %somehash){  # Same as bynumdesc
foreach $k (sort {$somehash{$a} cmp $somehash{$b}}
                   keys %somehash){            # Same as bystrvalue
```

Part IX

Appendix

Appendix A

Some examples of event-driven programming in Swing

A.1 Multicasting

We want two panels, with three buttons each labelled *Red*, *Blue* and *Yellow*. Clicking a button should change the colour of *both* panels to the appropriate colour.

Here, we make both panels listen to *all* six buttons. However, each panel has a reference only to the three buttons that are defined within it. Thus, we cannot directly use `getSource()` to decode the identity of a button from the other panel.

An alternative solution is to use the `ActionCommand` associated with a button. In the `ButtonPanel`, we set an `ActionCommand` with each button when we create it. In `ActionPerformed` we examine the `ActionCommand` of the button that was pressed to decide which colour to choose. We do not associate a listener with each button in the panel when we construct the panel. Instead, we provide a method `addListener` that adds a listener to all three buttons in the panel. This is invoked by the surrounding `ButtonFrame`, which defines two `ButtonPanels` and makes both of them listeners for each other.

ButtonPanel.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonPanel extends JPanel implements ActionListener{

    private JButton yellowButton;
    private JButton blueButton;
    private JButton redButton;
    public ButtonPanel(){
        yellowButton = new JButton("Yellow");
        blueButton = new JButton("Blue");
        redButton = new JButton("Red");

        yellowButton.setActionCommand("YELLOW");
        blueButton.setActionCommand("BLUE");
        redButton.setActionCommand("RED");

        add(yellowButton);
        add(blueButton);
        add(redButton);
    }

    public void actionPerformed(ActionEvent evt){
        Color color = getBackground();
        String cmd = evt.getActionCommand();   // Use ActionCommand to
                                               //  determine what to do

        if (cmd.equals("YELLOW")) color = Color.yellow;
        else if (cmd.equals("BLUE")) color = Color.blue;
        else if (cmd.equals("RED")) color = Color.red;

        setBackground(color);
        repaint();
    }

    public void addListener(ActionListener o){
        yellowButton.addActionListener(o);     // Add a commmon listener
        blueButton.addActionListener(o);       //   for all buttons in
        redButton.addActionListener(o);        //   this panel
    }
}
```

ButtonFrame.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


public class ButtonFrame extends JFrame implements WindowListener{
    private Container contentPane;
    private ButtonPanel b1, b2;

    public ButtonFrame(){
        setTitle("ButtonTest");
        setSize(300, 200);
        addWindowListener(this);

        b1 = new ButtonPanel();   // Create two button panels
        b2 = new ButtonPanel();

        b1.addListener(b1);       // Make each panel a listener for
        b1.addListener(b2);       //   both sets of buttons
        b2.addListener(b1);
        b2.addListener(b2);

        contentPane = this.getContentPane();

        contentPane.setLayout(new BorderLayout());  // Set layout to
        contentPane.add(b1,"North");                //   ensure that
        contentPane.add(b2,"South");                //   panels don't
                                                    //   overlap
    }

    public void windowClosing(WindowEvent e){
      System.exit(0);
    }
    public void windowActivated(WindowEvent e){}
    public void windowClosed(WindowEvent e){}
    public void windowDeactivated(WindowEvent e){}
    public void windowDeiconified(WindowEvent e){}
    public void windowIconified(WindowEvent e){}
    public void windowOpened(WindowEvent e){}

}
```

The main program that creates and displays a ButtonFrame is the same as before:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonTest
{  public static void main(String[] args)
```

```
   {  JFrame frame = new ButtonFrame();
      frame.show();
   }
}
```

A.2 Checkbox

A *checkbox* is a button with a one-bit state—a checkbox is either "selected" or "not selected". In Swing, the basic checkbox class is JCheckBox. Like a button, there is only one action that a user can perform on a checkbox—to click on it. Thus, in Java the listener for JCheckBox is the same as that for JButton—ActionListener. Clicking a checkbox toggles its state. We use the method isSelected() to determine the current state of a JCheckBox.

In CheckBoxPanel.java below, we define a JPanel with two checkboxes. The panel CheckBoxPanel serves as an ActionListener for both its checkboxes. When a checkbox is clicked, it examines the state of both checkboxes to determine which colour to paint the background.

As with ButtonPanel, we have to embed CheckBoxPanel in a JFrame, called CheckBoxFrame below, in order to display it. There is no real difference between the code for CheckBoxFrame and ButtonFrame which we have seen earlier.

Finally, we have a main program that creates and displays a CheckBoxFrame.

CheckBoxPanel.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


public class CheckBoxPanel extends JPanel implements ActionListener{

    private JCheckBox redBox;
    private JCheckBox blueBox;

    public CheckBoxPanel(){

        redBox = new JCheckBox("Red");
        blueBox = new JCheckBox("Blue");

        redBox.addActionListener(this);
        blueBox.addActionListener(this);

        redBox.setSelected(false);
        blueBox.setSelected(false);

        add(redBox);
        add(blueBox);

    }

    public void actionPerformed(ActionEvent evt){

        Color color = getBackground();

        if (blueBox.isSelected()) color = Color.blue;
        if (redBox.isSelected()) color = Color.red;
        if (blueBox.isSelected() && redBox.isSelected()) color = Color.green;

        setBackground(color);
        repaint();
    }
}
```

CheckBoxFrame.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


public class CheckBoxFrame extends JFrame implements WindowListener{
    private Container contentPane;

    public CheckBoxFrame(){
        setTitle("BoxTest");
        setSize(300, 200);
        addWindowListener(this);

        contentPane = this.getContentPane();

        contentPane.add(new CheckBoxPanel());


    }

    public void windowClosing(WindowEvent e){
        System.exit(0);
    }

    public void windowActivated(WindowEvent e){}

    public void windowClosed(WindowEvent e){}

    public void windowDeactivated(WindowEvent e){}

    public void windowDeiconified(WindowEvent e){}

    public void windowIconified(WindowEvent e){}

    public void windowOpened(WindowEvent e){}

}
```

CheckBoxTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CheckBoxTest
{  public static void main(String[] args)
   {  JFrame frame = new CheckBoxFrame();
      frame.show();
   }
}
```