

# Lazy Functional Programming in Java

Anthony H. Dekker

Defence Science and Technology Organisation  
Department of Defence, Canberra ACT 2600

dekker@ACM.org

## Abstract

In this paper, we show how lazy functional programming techniques can be used within the Java programming language. We provide Java implementations of classic examples of lazy lists, such as the Sieve of Eratosthenes, the Eight Queens Problem, and natural-language parsing. We discuss how well these implementations succeed, compared to their original counterparts. We also point out the potential synergy between adding lazy techniques to Java, and adding generic types. The examples we provide would be suitable for teaching functional programming concepts in the context of a Java-based syllabus.

*Keywords:* lazy lists, functional programming, Java.

## 1 Introduction

During the 1980s, advocates of functional programming (Bird & Wadler 1988, Reade 1989) demonstrated that functional programming languages provided several important aids to the programmer:

1. **Garbage collection** allowed the creation and destruction of complex data structures without the programmer needing to explicitly manage memory.
2. **Static strong typing** allowed a large class of incorrect programs to be identified by the compiler, thus saving needless debugging effort.
3. **Polymorphism** allowed generic functions to be defined while retaining static strong typing (e.g. a function to find the first element of a list, regardless of the type of list element).
4. **Higher-order functions** allowed generalisations of operations to be defined (e.g. a list-sort operation, given an arbitrary comparison function).
5. **Lazy evaluation** allowed program termination to be decoupled from definition, thus avoiding unnecessary computation. Lazy evaluation also made it possible to find either **one** solution to a problem, or **all** solutions, using the same problem-solving code.

The Java™ programming language (Sun Microsystems 2005) brought the benefits of garbage collection and static strong typing to a wider audience, in the context of an object-oriented programming language.

Without a polymorphic type system, Java could define generic data structures (such as `Vector`) and generic operations only by using the `Object` type and performing run-time typecasting. However, this is being remedied, and generic types are being added to the Java programming language (Java Community Process 2005).

Java provided an alternative to higher-order functions by using **interfaces** which specified a method name. Parameter objects could then implement the method in a range of different ways. For example, a sorting routine could require a parameter to implement the `Comparable` interface, containing a `compareTo()` method. In the extreme case, parameter objects serve no purpose other than as “place holders” for the given method—as in event handlers.

An example of a “place holder” is the following interface—objects implementing it serve no purpose except to implement the `isOK()` method:

```
public interface Filter {  
    public boolean isOK (Object item);  
}
```

In this paper, we will show how the fifth functional programming technique—lazy evaluation—can be used within standard Java. The examples provided in the paper would be suitable for teaching functional programming concepts in the context of a Java-based syllabus.

## 2 Processes

The concept of a lazy list can be represented in Java as a process which returns objects either forever, or until no more are left, that is:

```
public interface Process {  
    public Object nextElement ()  
        throws NoSuchElementException;  
}
```

This is very similar to the standard Enumeration interface. Indeed, we feel that Process should be a superclass of Enumeration. However, Process does not have a method to explicitly test for the presence of the next element. Since Java provides excellent exception handling, it is easier to rely on exceptions in order to recognise the end of the lazy list—and doing so simplifies programming, as we will see later.

Corresponding to the empty list, it is easy to define an implementation of the Process interface which contains no elements, i.e. it always throws a NoSuchElementException:

```
public class NullProcess
    implements Process {

    public NullProcess () {
    }

    public Object nextElement ()
        throws NoSuchElementException {
        throw new NoSuchElementException();
    }
}
```

Similarly, a one-element list corresponds to a Process that returns an object only the first time that nextElement() is called:

```
public class SingleProcess
    implements Process {

    private Object item;

    public SingleProcess (Object item) {
        this.item = item;
    }

    public Object nextElement ()
        throws NoSuchElementException {
        if (item == null) throw
            new NoSuchElementException();
        else {
            Object temp = item;
            item = null;
            return temp;
        }
    }
}
```

List concatenation is implemented by Process that contains two subprocesses. Objects are fetched from the first subprocess until it is finished, and then from the second. This is a common design pattern for processes, being a kind of sum:

```
public class ConcatProcess
    implements Process {

    private Process firstP;
    private Process secondP;
    private boolean firstFinished;

    public ConcatProcess
        (Process a, Process b) {
        firstP = a;
        secondP = b;
        firstFinished = false;
    }

    public Object nextElement ()
        throws NoSuchElementException {
        if (firstFinished) {
            return secondP.nextElement ();
        } else {
            try {
                return firstP.nextElement ();
            } catch
                (NoSuchElementException ex) {
                firstFinished = true;
                return secondP.nextElement ();
            }
        }
    }
}
```

The important operation of **list filtering** can be implemented by process which uses a Filter object (as defined above) as a “placeholder” for a predicate on list elements. The nextElement() method then returns only elements satisfying the predicate (this is an example of a case where the Process interface is easier to implement than Enumeration would be):

```
public class FilterProcess
    implements Process {

    private Process process;
    private Filter filter;

    public FilterProcess (Process p,
        Filter f) {
        process = p;
        filter = f;
    }

    public Object nextElement ()
        throws NoSuchElementException {

        while (true) {
            Object x = process.nextElement();
            if (filter.isOK (x)) return x;
        }
    }
}
```

Another simple infinite list is a sequence of integers starting from a given value. This is also easily specified:

```
public class CountProcess
    implements Process {
    private int upto = 0;

    public CountProcess (int from) {
        upto = from;
    }

    public Object nextElement () {
        return new Integer (upto ++);
    }
}
```

A classic example of an infinite list is the **Sieve of Eratosthenes** for finding prime numbers (Bird & Wadler 1988, Reade 1989). The definition of this as a Process involves beginning with a CountProcess counting from 2, and, every time a prime number is generated, adding a FilterProcess in front of the sieve:

```
public class PrimeSieve
    implements Process {
    private Process p;

    public PrimeSieve () {
        p = new CountProcess (2);
    }

    public Object nextElement ()
        throws NoSuchElementException {
        Integer prime
            = (Integer) p.nextElement();
        Filter f = new NotMultFilter
            (prime.intValue ());
        p = new FilterProcess (p, f);
        return prime;
    }

    public static void main(String[] a){
        Process p = new PrimeSieve ();
        while (true) {
            try {
                Integer prime
                    = (Integer) p.nextElement ();
                System.out.println (prime);
            } catch
                (NoSuchElementException ex){
                break;
            }
        }
    }
}
```

The main method here prints out all the primes in an infinite loop (increasingly slowly, of course). After the numbers 2 and 3 have been generated, the sieve has the structure shown in Figure 1:

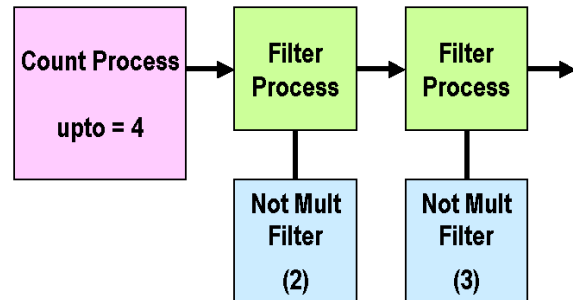


Figure 1: PrimeSieve Operation

The Filter object used by PrimeSieve simply checks for integers that are not multiples of a given number:

```
public class NotMultFilter
    implements Filter {
    private int number = 0;

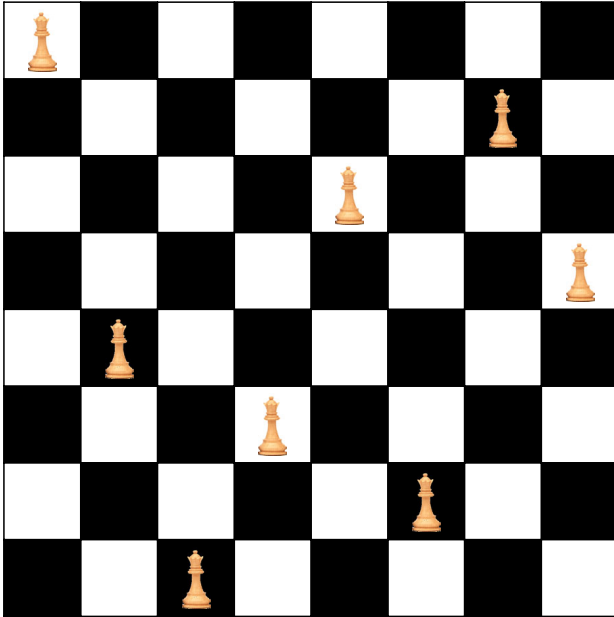
    public NotMultFilter (int n) {
        number = n;
    }

    public boolean isOK (Object item) {
        if (item instanceof Integer) {
            int value
                = ((Integer) item).intValue ();
            return value % number != 0;
        } else {
            return false;
        }
    }
}
```

### 3 The Eight Queens Problem

Another classic example of the use of infinite lists is the Eight Queens Problem (Bird & Wadler 1989). This involves placing 8 queens on a chessboard (or  $N$  queens on a generalised chessboard), so that none are in the same row, column, or diagonal. Figure 2 is one of 92 possible solutions for the case of  $N=8$ .

Solving this kind of problem requires backtracking, and lazy evaluation makes it easier to program this, using the “list of successes” technique (Bird & Wadler 1989). Lazy evaluation also simplifies the code by separating the solution technique from the termination condition. This allows the same code to find one solution, all solutions, or (for example) the first three solutions.



**Figure 2: Eight Queens Solution**

Solving the problem requires a `Queen` class that represents a (possibly partial) solution, with queens placed in the first `position` columns. The `Queen` class contains a `toString()` method, a test to see whether it is safe to add a new queen in the next column, and a `clone()` method so that multiple solutions can be returned as non-interfering objects:

```
public class Queen
    implements Cloneable {

    private int position = 0;
    private int size = 0;
    private int [] columns = null;

    private Queen (int pos, int [] c) {
        size = c.length;
        position = pos;
        columns = new int [c.length];

        for (int i = 0; i < c.length; i++){
            columns [i] = c [i];
        }
    }

    protected Object clone () {
        return new Queen(position, columns);
    }

    public Queen (int size) {
        this.size = size;
        columns = new int [size];
    }
}
```

```
public String toString () {
    String s = "[";
    for (int i = 0; i < position; i++){
        s +=
            (i==0 ? "" : ",") + columns[i];
    }
    return s + "]";
}

public int getRowNum (int colNum) {
    return columns [colNum];
}

public boolean isComplete () {
    return (position == size);
}

public void addColumn (int row) {
    columns [position] = row;
    position ++;
}

public void resetPosition (int p) {
    position = p;
}

public boolean safeToAdd (int row) {
    for (int i = 0; i < position; i++){
        int j = columns [i];
        if (j == row ||
            i+j == position+row ||
            i-j == position-row)
            return false;
    }
    return true;
}
}
```

The process to solve the Eight Queens problem is mostly setup code, with the `makeQP()` static method setting up  $N$  nested processes in a chain, each responsible for placement of queens in a specified column. The chain begins with a `SingleProcess` that returns an empty board (as a partial solution to the problem of placing 0 queens).

The `nextElement()` method here uses another useful design pattern, corresponding to a kind of product. It contains an infinite loop with the body in two parts:

- (a) to fetch another partial solution from the nested process, if one is required, and
- (b) to place a Queen on the different squares in the column that the process controls, keeping track of the square it is up to with the variable `step`.

This behaves like a product in that the partial solutions produced by the nested process are each combined with the positions 0...7 in the current column (if these are legal):

```
public class QueenProcess
    implements Process {
    private int size = 0;
    private Process nextProcess = null;
    private Queen current = null;
    private int step = 0;

    private QueenProcess(int s, int lev) {
        size = s;
        nextProcess = makeQP (s, lev-1);
    }

    private static Process makeQP
        (int s, int lev) {
        if (level == 0) {
            Queen q = new Queen (s);
            return new SingleProcess (q);
        } else {
            return new QueenProcess(s, lev);
        }
    }

    public static Process makeQP(int s){
        return makeQP (s, s);
    }

    public Object nextElement ()
        throws NoSuchElementException {
        while (true) {           /* part A */
            if (current == null) {
                step = 0;
                current = (Queen)
                    nextProcess.nextElement ();
            }

            if (current.safeToAdd (step)) {
                Queen q           /* part B */
                    = (Queen) current.clone();
                q.addColumn (step);
                step ++;
                if (step >= size)
                    current = null;
                return q;
            } else {
                step ++;
                if (step >= size)
                    current = null;
            }
        }
    }
}
```

```
public static void main(String[] a){
    Process p = makeQP (8);
    int n = 0;

    while (true) {
        try {
            Queen q
                = (Queen) p.nextElement();
            System.out.println (q);
            n ++;
        } catch(NoSuchElementException e){
            break;
        }
    }
    System.out.println
        (n + " solutions");
}
```

The main method here prints out 92 solutions, the first of which is [0,4,7,5,2,6,1,3], the solution shown in Figure 2.

#### 4 Remembering the Past

The Process interface only defines a lazy list that is consumed as fast as it is produced. If previous elements need to be remembered, then the following class defines a linked list that can “unwind” a Process as required. Actually, it is currently defined using Enumeration. However, if Process was a superclass of Enumeration, as we suggest, then substituting Process for Enumeration would provide a class that worked for both alternatives.

Each EnumerationList object contains either an (as yet) unevaluated Enumeration object, or a value/next pair as in a standard linked list:

```
public class EnumerationList {
    private Enumeration enum = null;
    private Object value = null;
    private EnumerationList next = null;
    private boolean empty = false;

    public EnumerationList
        (Enumeration e) {
        enum = e;
    }

    public Object getValue () {
        check ();
        return value;
    }

    public EnumerationList getNext () {
        check ();
        return next;
    }
}
```

```

public boolean isEmpty () {
    check ();
    return empty;
}

private void check () {
    if (enum == null) {
        return;
    } else {
        try {
            value = enum.nextElement ();
            next
                = new EnumerationList (enum);
            enum = null;
            empty = false;
        } catch
            (NoSuchElementException ex){
                enum = null;
                empty = true;
            }
    }
}
}
}

```

## 5 Backtracking Parsers

Using the Eight Queens problem, we have shown how lazy evaluation allows a backtracking solution to be implemented in Java. Backtracking is also important in natural-language parsing, where the ambiguity of natural language means that multiple parses are possible. A natural-language parser is therefore a kind of `Process`, which returns a lazy list of the possible parses.

Because of backtracking, the `Parser` must accept tokens as an `EnumerationList`, so that it can redo parsing from any given point. For the same reason, parsers need to be `Cloneable`, so that copies of a parser can process different positions in the input.

The `Parser` class provides a `nextParse ()` method, which is essentially the same as `nextElement ()`:

```

public abstract class Parser
    implements Process, Cloneable {
    protected boolean finished = false;
    protected EnumerationList tokens
        = null;

    protected Object clone () {
        try {
            return super.clone ();
        } catch
            (CloneNotSupportedException e){
                return null;
            }
    }
}

```

```

public void setTokens
    (EnumerationList t) {
    tokens = t;
    finished = false;
}

public abstract ParseRes nextParse()
    throws NoSuchElementException;

public Object nextElement ()
    throws NoSuchElementException {
    return nextParse ();
}

public Vector allParses () {
    Vector all = new Vector ();
    while (true) {
        try {
            all.addElement (nextParse ());
        } catch
            (NoSuchElementException ex){
                return all;
            }
    }
}
}
}

```

Parsers return `ParseRes` objects, which contain the value resulting from a parse, and any unconsumed input:

```

public class ParseRes {
    public EnumerationList remainder;
    public Object value;

    public ParseRes (Object v,
        EnumerationList rem) {
        value = v;
        remainder = rem;
    }

    public boolean isComplete () {
        return remainder.isEmpty ();
    }

    public String toString () {
        return value
            + (isComplete () ? "" : "...");
    }
}

```

A simple example of a `Parser` is one which recognises a single token from the input list, provided that it occurs in an array of legal tokens:

```

public class MatchParser extends Parser{
    private Object [] ok;

    public MatchParser (Object [] ok) {
        this.ok = ok;
    }

    private boolean legal (Object item){
        if (ok == null || item == null) {
            return false;
        } else {
            for (int i=0; i<ok.length; i++){
                if (item.equals (ok [i])) {
                    return true;
                }
            }
            return false;
        }
    }

    public ParseRes nextParse ()
        throws NoSuchElementException {
        if (finished) throw
            new NoSuchElementException();
        else {
            finished = true;
            if (tokens.isEmpty ()) throw
                new NoSuchElementException();
            else {
                Object first
                    = tokens.getValue ();
                if (legal (first))
                    return new ParseRes
                        (first, tokens.getNext());
                else throw
                    new NoSuchElementException();
            }
        }
    }
}

```

We can build up a natural-language parser by defining a series of operators that allow simple parses to be combined to form more complex ones. A complete parser for any natural language is beyond the scope of this paper, so we will content ourselves with defining a limited selection of operators, and a very simple example natural-language parser.

The first parser operator is a sequencing operator which applies first one parser to the input, and then another. Since parsers produce multiple results, this will require the same product-like design pattern used in `QueenProcess`:

```

public class SeqParser extends Parser {
    private Parser firstP;
    private Parser secondP;
    private ParseRes firstRes;

    public SeqParser(Parser a,Parser b){
        firstP = (Parser) a.clone ();
        secondP = b;
    }

    protected Object clone () {
        Parser a = (Parser)firstP.clone();
        Parser b = (Parser)secondP.clone();
        return new SeqParser (a, b);
    }

    public void setTokens
        (EnumerationList t) {
        tokens = t;
        firstRes = null;
        firstP.setTokens (tokens);
    }

    public ParseRes nextParse ()
        throws NoSuchElementException {
        while (true) {
            if (firstRes == null) {
                firstRes = firstP.nextParse();
                secondP.setTokens
                    (firstRes.remainder);
            }
            try {
                ParseRes sndRes
                    = secondP.nextParse ();
                Vector v = new Vector ();
                v.addElement (firstRes.value);
                v.addElement (sndRes.value);
                ParseRes res = new ParseRes(v,
                    secondRes.remainder);
                return res;
            } catch
                (NoSuchElementException ex){
                firstRes = null;
            }
        }
    }
}

```

We can use this to construct a very simple example parser, using `MatchParser` and `SeqParser`:

```

Parser noun = new MatchParser (new
    String [] {"John", "Mary", "Peter"});

```

```

Parser verb = new MatchParser (new
    String [] {"likes", "hates"});
Parser sentence = new SeqParser (
    (noun, new SeqParser (verb, noun));

```

Feeding in as a token list an EnumerationList from a StringTokenizer for “Mary likes Peter”:

```

EnumerationList t = new EnumerationList
    (new StringTokenizer (
        "Mary likes Peter"));

```

We obtain the single parse [Mary, [likes, Peter]], matching the sentence structure we specified.

The other important operator on parsers is an either-or operator, which follows the “sum” design pattern:

```

public class AltParser extends Parser {
    private Parser firstP;
    private Parser secondP;
    private boolean fstFinished = false;

    public AltParser(Parser a, Parser b) {
        firstP = a;
        secondP = b;
    }

    protected Object clone () {
        Parser a = (Parser)firstP.clone();
        Parser b = (Parser)secondP.clone();
        return new AltParser (a, b);
    }

    public void setTokens
        (EnumerationList t) {
        tokens = t;
        fstFinished = false;
        firstP.setTokens (t);
    }

    public ParseRes nextParse ()
        throws NoSuchElementException {
        if (fstFinished) return
            secondP.nextParse ();
        else {
            try {
                return firstP.nextParse ();
            } catch
                (NoSuchElementException ex) {
                fstFinished = true;
                secondP.setTokens (tokens);
                return secondP.nextParse ();
            }
        }
    }
}

```

Using AltParser, we can define a slightly more complex, but still “toy,” natural-language parser:

```

Parser noun = new MatchParser
    (new String [] {"fruit", "flies",
        "banana"});
Parser verb = new MatchParser
    (new String [] {"like", "flies"});
Parser det = new MatchParser
    (new String [] {"a", "the"});
Parser adj = new MatchParser
    (new String [] {"fruit", "nice"});
Parser nounP1 = new AltParser (noun,
    new SequenceParser (adj, noun));
Parser nounP = new AltParser (nounP1,
    new SequenceParser (det, nounP1));
Parser like = new MatchParser
    (new String [] {"like"});
Parser verbMod = new SequenceParser
    (like, nounP);
Parser verbP = new AltParser (verb,
    new SequenceParser (verb, verbMod));
Parser s1 = new SequenceParser
    (nounP, verbP);
Parser s2 = new SequenceParser (nounP,
    new SequenceParser (verbP, nounP));
Parser sentence
    = new AltParser (s1, s2);

```

Feeding in as a token list an EnumerationList from a StringTokenizer for “fruit flies like a nice banana” we get four alternative parses (two of which are partial):

- [fruit, flies]...
- [fruit, [flies, [like, [a, [nice, banana]]]]]
- [[fruit, flies], like]...
- [[fruit, flies], [like, [a, [nice, banana]]]]

The alternatives are possible because either “flies” or “like” can be verb of the sentence, and because there are two forms of sentence allowed.

## 6 Transformations

A realistic parser also requires an operator to manipulate parse trees. Such an operator is also easy to define, using the same basic techniques:

```

public class TransformParser
    extends Parser {
    private Transform transform;
    private Parser parser;

    public TransformParser
        (Parser p, Transform t) {
        parser = p;
        transform = t;
    }

    protected Object clone () {
        Parser p = (Parser)parser.clone();
        return new TransformParser
            (p, transform);
    }

    public void setTokens
        (EnumerationList t) {
        parser.setTokens (t);
    }

    public ParseRes nextParse ()
        throws NoSuchElementException {
        ParseRes res = parser.nextParse();
        Object y = transform.transform
            (res.value);
        return new ParseRes
            (y, res.remainder);
    }
}

```

This operator on parsers requires a function input, much like `FilterProcess`, and using an interface in exactly the same way:

```

public interface Transform {
    public Object transform (Object x);
}

```

Indeed, having defined this interface, we can also use it to implement a “map” operator on lazy lists:

```

public class TransformProcess
    implements Process {
    private Process process;
    private Transform transform;

    public TransformProcess
        (Process p, Transform t) {
        process = p;
        transform = t;
    }
}

```

```

    public Object nextElement ()
        throws NoSuchElementException {
        Object x = process.nextElement ();
        return transform.transform (x);
    }
}

```

Composition of Transform objects is also possible:

```

public class ComposeTransform
    implements Transform {
    private Transform f;
    private Transform g;

    public ComposeTransform
        (Transform f, Transform g) {
        this.f = f;
        this.g = g;
    }

    public Object transform (Object x) {
        Object y = g.transform (x);
        return f.transform (y);
    }
}

```

However, correct typing of Transform objects, and particularly their composition, requires generic types.

## 7 Discussion

We have demonstrated that lazy functional programming techniques can be used with the Java programming language, and have shown how classic examples such as the Sieve of Eratosthenes, the Eight Queens problem, and backtracking parsers can be implemented in Java.

However, Java lacks the conciseness of functional programming languages. Measured by lines of code, the Java solutions for the Sieve of Eratosthenes and the Eight Queens Problem are approximately 10 times larger than solutions in functional programming languages (Bird & Wadler 1988, Reade 1989). Even though each line of Java code is somewhat simpler than those in the functional programming language solutions, this still represents a significant increase in complexity. This increase in complexity is the inevitable result of using a procedural, rather than a declarative, programming style.

Our recommended `Process` class also only implements lazy lists that are consumed as fast as they are produced. Remembering lists elements for future use requires explicitly coding the “unwinding” of a lazy list, as in `EnumerationList`. This represents a significant

decrease in flexibility, but is inevitable when using a non-functional language like Java.

The `Process` class relies on using exceptions to recognise the end of a lazy list. Although Java provides excellent exception-handling facilities, programmers are often reluctant to use them. We would suggest that exceptions not be viewed as exceptional: that they be viewed as a natural part of programming, just as `fail` is used in the Prolog programming language (Clocksin & Mellish 1984). However, exceptions need not be restricted to recording failure: subclasses of `Exception` can also be used to return results.

The examples that we have provided also cry out for the implementation of generic types. For example, `PrimeSieve` is intended to produce a `Process` of `Integer` objects only, and so `NotMultFilter` should not need to typecast items. Similarly, `QueenProcess` is intended to produce a `Process` of `Queen` objects only, and so contains three unnecessary typecasts. The `Transform` interface, as used in `TransformParser` and `TransformProcess` also requires generic types in order to be typechecked properly.

Nevertheless, we have shown that the techniques of lazy functional programming can be used within the Java

programming language. Such techniques are of considerable practical benefit when backtracking needs to be used for problem-solving. The examples we have provided would also be suitable for teaching functional programming concepts in the context of a Java-based syllabus.

## 8 References

- Bird, R. & Wadler, P. (1988), *Introduction to Functional Programming*, Prentice Hall.
- Clocksin, W.F. & Mellish, C.S. (1984), *Programming in Prolog*, 2<sup>nd</sup> edition, Springer-Verlag.
- Java Community Process (2005), Java Specification Request 14: add generic types to the Java<sup>TM</sup> Programming Language: <http://jcp.org/en/jsr/detail?id=014>
- Reade, C. (1989), *Elements of Functional Programming*, Addison-Wesley.
- Sun Microsystems (2005), Java Technology home page: <http://java.sun.com/>