JFun: Functional Programming in Java

Tekin Meriçli, Peng Bi
Department of Computer Sciences
The University of Texas at Austin
{tmericli,pengbi}@cs.utexas.edu

Abstract

A function is a good way of specifying a computation since in each computation the result depends in a certain way on the parameters, and using functions makes a program modular and well-structured. In order to reduce the development effort and future programming costs caused by bugs and maintenance problems, writing well-structured and modular programs has become crucial as software becomes more and more complex. Since modularity is the key to successful programming, functional languages are vitally important to the real world. In this paper we try to show that writing programs in an object-oriented language by using functional programming concepts is possible. As examples, list and tree manipulation functions, numerical integration and differentiation, and alpha-beta heuristic which is an algorithm from Artificial Intelligence used in game-playing programs are implemented in Java programming language using functional programming concepts.

1 Introduction

The first computers were built during the 40s, and huge connection boards were used to "program" those very first models. Soon the programs were started to be stored in the memory of the computer, introducing the first programming languages. It was obvious to have the programming language resemble the architecture of the computer as close as possible since the computer use was very expensive in those days. A computer mainly consists of a central processing unit and a memory; therefore, a program consisted of instructions to modify the memory, executed by the processing unit. With that the imperative programming style arose. Imperative programming languages, like Pascal and C, are characterized by the existence of assignments, executed sequentially.

Of course there were some methods to solve problems before the invention of computers, one of which is pure math. In math, for at least the last four hundred years, functions have played a central role. Functions express the connection between parameters (the *input*) and the result (the *output*) of certain processes. A function is a good way of specifying a computation since in each computation the result depends in a certain way on the parameters. This is the basis of the *functional programming* style. In functional programming, a *program* consists of the definition of one or more functions. During the "execution" of a program, some parameters are provided to the function, and the function is expected to calculate and return the result [2].

In this paper, we implement some essential functional programming elements using Java, which is a structured object-oriented programming language, to have a deeper understanding in the difference between programming in a functional way and in a objected-oriented structural way. The rest of the paper is organized as follows. Section 2 gives some brief background information on functional programming and object-oriented programming, and provides a comparison between functional programming and imperative programming in general. Design stage of the project is explained in detail in Section 3. Section 4 gives Java implementation details of some of the examples given in [1] as well as some general examples of using functional programming concepts in object-oriented programming in order to provide further understanding of the subject. Finally, Section 5 summarizes the work.

2 Background

This section gives some brief information about functional programming and object-oriented programming, and provides a comparison of functional programming and imperative programming.

2.1 Functional Programming

C, Java, Pascal, Ada, and so on, are all *imperative languages* in the sense that they consist of a sequence of commands, which are executed strictly one after the other. On the other hand, Haskell, for example, is a *functional language.* A functional program is a single expression, which is executed by evaluating the expression.

Functional programming supports functions as first-class objects, sometimes called *closures*, or *functor objects.* Essentially, closures are objects that act as functions and can be operated upon as objects. Similarly, FP languages support higher order functions. A higher order function is able to take another function (indirectly, an expression) as its input argument, and in some cases it may even return a function as its output argument. These two constructs together allow for elegant ways to modularize programs, which is one of the biggest advantages of using FP.

Anyone who has used a spreadsheet has experience of functional programming since spreadsheets are one common example of the use of functional programming. In a spreadsheet, one specifies the value of each cell in terms of the values of other cells. The focus is on what is to be computed, not how it should be computed. For example:

- no need to specify the order in which the cells should be calculated - instead we take it for granted that the spreadsheet will compute cells in an order which respects their dependencies.

- no need to tell the spreadsheet how to allocate its memory - rather, we expect it to present us with an apparently infinite plane of cells, and to allocate memory only to those cells which are actually in use.

- for the most part, we only specify the value of a cell by an expression (whose parts can be evaluated in any order), rather by a sequence of commands which computes its value.

An interesting consequence of the spreadsheet's unspecified order of re-calculation is that the notion of assignment is not very useful. After all, if you don not know exactly when an assignment will happen, you can not make much use of it. This contrasts strongly with programs in conventional languages like C, which consist essentially of a carefully-specified sequence of assignments, or Java, in which the ordering of method calls is crucial to the meaning of a program. This focus on the high-level "what" rather than the low-level "how" is a distinguishing characteristic of functional programming languages.

Another well-known nearly-functional language is the standard database query language SQL. An SQL query is an expression involving projections, selections, joins and so forth. The query says what relation should be computed, without saying how it should be computed. Indeed, the query can be evaluated in any convenient order. SQL implementations often perform extensive query optimization which (among other things) figures out the best order in which to evaluate the expression.

So far we explained the characteristic differences that separate functional programming from imperative programming. Before proceeding with the benefits of functional programming, we will provide an example of "quick-sort" program written in Haskell, in order to give the idea of what a functional program looks like.

```
qsort [] = []
qsort (x:xs) = qsort (filter (< x) xs) ++ [x]
++ qsort (filter (>= x) xs)
```

2.1.1 Benefits of Functional Programming

As seen in the quicksort example, functional programs are much shorter (two to ten times) in general. Therefore, one benefit of functional programs is being much more concise compared to their imperative counterparts. Also, functional programs are often easier to understand. For example, the first line of the quicksort program reads: "The result of sorting an empty list ([]) is an empty list". The second line reads: "To sort a list whose first element is x and the rest of which is called xs, sort the elements of xs that are less than x, sort the elements of xs that are greater than or equal to x, and concatenate (++) the results with x sandwiched in the middle."

Safety is one of the most important properties of a functional program. Most functional languages are *strongly typed.* Therefore, a huge class of easy-to-make errors is eliminated at compile time. In particular, strong typing means no core dumps; that is, there is simply no possibility of treating an integer as a pointer, or following a null pointer.

Polymorphism and code re-use can be listed as the other important benefits of functional programming. For example, the quicksort program given in the previous section will not only sort lists of integers, but also lists of floating point numbers, lists of characters, lists of lists; indeed, it will sort lists of anything which can be compared by the less-than and greater-than operations.

Although there are a lot more, one last benefit that should be mentioned is the support of powerful *abstractions.* In general, functional languages offer powerful new ways to encapsulate abstractions. An abstraction allows you to define an object whose internal workings are hidden; a C procedure, for example, is an abstraction. Abstractions are the key to building modular, maintainable programs. One powerful abstraction mechanism available in functional languages is the *higher-order function.* In Haskell, for example, a function can freely be passed to other functions, returned as the result of a function, stored in a data structure, and so on. It turns out that the judicious use of higher-order functions can substantially improve the structure and modularity of many programs [3].

2.2 Object-Oriented Programming

In short, object-oriented programming is a type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions or methods as in the OOP jargon) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify.

2.2.1 Java

Java is an object-oriented programming language developed initially by James Gosling and colleagues at Sun Microsystems. The language, initially called Oak (named after the oak trees outside Gosling's office), was intended to replace C++, although the feature set better resembles that of Objective C. Unlike conventional languages which are generally designed to be compiled to native code, Java is compiled to a bytecode which is then run (generally using JIT compilation) by a Java virtual machine. The syntax of Java is largely derived from C++. But unlike C++, which combines the syntax for structured, generic, and object-oriented programming, Java was built from the ground up to be fully object-oriented. Everything in Java is an object (with a few exceptions), and everything in Java is written inside a class [4]. Here is an illustrative example:

Listing 1: Example of the classical "Hello World" program written in Java

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

2.2.2 Generics in Java

The need for generic types stems from the implementation and use of collections, like those in the Java collection framework. Typically, the implementation of a collection of objects is independent of the type of the objects that the collection maintains. Therefore, it does not make sense to re-implement the same data structure over and over again, just because it will hold different types of elements. Instead, the goal is to have a single implementation of the collection and use it to hold elements of different types. In other words, rather than implementing a class IntList and StringList for holding integral values and strings, respectively, we want to have one generic implementation List that can be used in either case.

Java is a strongly typed language. When programming with Java, at compile time, you expect to know if you pass a wrong type of parameter to a method. For instance, if you define

```
Dog aDog = aBookReference; // ERROR
```

where, aBookReference is a reference of type Book, which is not related to Dog, you would get a compilation error. Unfortunately though, when Java was introduced, this was not carried through fully into the Collections library. So, for instance, you can write

```
Vector vec = new Vector();
vec.add(``hello'');
vec.add(new Dog());
```

There is no control on what type of object you place into the Vector during compilation,; however, this causes the program to run incorrectly.

Being very similar to C++ templates, generics allow the programmer to abstract over types. The most common examples are container types, such as those in the Collection hierarchy. Here is a typical usage of that sort:

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. Here is a version of the program fragment given above using generics:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1
myIntList.add(new Integer(0)); //2
Integer x = myIntList.iterator().next(); // 3
```

Notice the type declaration for the variable myIntList. It specifies that this is not just an arbitrary List, but a List of Integer, written List<Integer>. We say that List is a generic interface that takes a type parameter - in this case, Integer. We also specify a type parameter when creating the list object [5].

2.3 Functional Programming vs. Imperative Programming

Although the comparison of functional programming and imperative programming is provided implicitly in the preceding sections, this section summarizes the concept.

Functional programming appears to be missing several constructs often (though incorrectly) considered essential to an imperative language such as C or Pascal. For example, in strict functional programming, there is no explicit memory allocation and no explicit variable assignment. However, these operations occur automatically when a function is invoked: memory allocation occurs to create space for the parameters and the return value, and assignment occurs to copy the parameters into this newly allocated space and to copy the return value back into the calling function. Both operations can only occur on function entry and exit, so side effects of function evaluation are eliminated. By disallowing side effects in functions, the language provides referential transparency which ensures that the result of a function will be the same for a given set of parameters no matter where or when it is evaluated. Referential transparency greatly eases both the task of proving program correctness and the task of automatically identifying independent computations for parallel execution.

Iteration (looping), another imperative programming construct, is accomplished through the more general functional construct of recursion. Recursive functions invoke themselves, allowing an operation to be performed over and over. In fact, it can be proven that iteration is equivalent to a special type of recursion called tail recursion. Recursion in functional programming can take many forms and is in general a more powerful technique than iteration. For this reason, almost all imperative languages also support

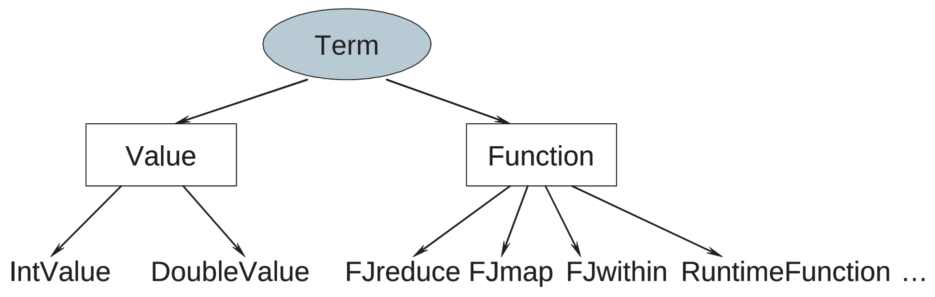

Figure 1: class hierarchy

it (with FORTRAN 77 and COBOL, before 2002, as notable exceptions).

Functional programming often depends heavily on recursion. The Scheme programming language even requires certain types of recursion (tail recursion) to be recognized and automatically optimized by a compiler.

3 Design

Essentially, because of the major distinction between Java and a functional programming language, we need to implement a mini functional compiler in order to take advantage of the features. The compiler has to

- understand the functional terms and have corresponding data structures to represent them internally in Java
- be able to evaluate the terms in the functional manner, and support lazy evaluation.

3.1 Types and Representation

Roughly speaking, the functional language we implement only needs two types: primitive type and function type. Primitive type includes `Nil`, integers and floating point numbers. The function type includes all functions, such as `cons`, `sum`, `reduce`, etc. The simplicity in the type system of JFun is different from an OOP language like Java, in which many complicated types can be defined, while the complicated types in JFun are expressed as functions, which are usually defined recursively.

The class hierarchy is showed in Figure 1. Class `Term` is the general representation, which is an abstract class. A term may be either a value of some primitive type, or a function, optionally with a set of parameters, each of which is also an instance of `Term`.

In Listing 2, there are some examples of how to create JFun terms in Java.

Listing 2: Examples of creating terms in Java

```
// integer 1
Term t = new IntValue(1);
// cons 1 Nil
Term con = new FJcons(t, new Value(null));
// cons 10 (cons 1 Nil)
Term con2 = new FJcons(new Value(10), con);
```

Note that each instance of `Term` class is *immutable*, meaning once the instance is created, the parameters of the function, or the value of the term are not allowed to be changed. This is consistent with the philosophy of functional programming languages, and makes the evaluation process much easier. For example, when we want to apply a parameter `p` to a function `f` by calling `f.addParameter(p)`, a new instance of the function object is exactly cloned from `f` before attaching `p`. More detailed information can be obtained from the Java source code in `Function.java`.

Of course, it could be possible to use Java generics in the implementation of the examples but this time we needed to declare each type we used, mainly `Value` and `Function`, as separate individual classes instead of extending the base class Term for creating those types. In that way, we would not need to use type-casting when, for example, reading a parameter using `getParam()` and want to make sure that the parameter is a function.

```
Value v = (Value)getParam(0);
Function f = (Function)getParam(1);
```

If we had used generics, the base type definition would be `Term<subtype>`, where `subtype` is either a `Value` or a `Function`. However, we solved that problem in our current implementation by using two different functions called `isFunction()`, which return `true` when a `Term` is a `Function`, and `isValue()`, which return `true` when a `Term` is a `Value`.

3.2 Parsing

One may notice that although we can create all JFun terms by using the Java syntax, the process is very tedious and might be error-prone. For instance, each value and each function need to be created with a `new` statement.

Therefore, we have designed a parsing algorithm to create terms in a flexible and concise way. The goal is to be able to parse a term string and create the object that represents the term automatically. Listing 3 gives the pseudo code for the parsing algorithm.

Listing 3: parsing algorithm

```
public static Term createTerm(String s, Term [] variables)
//input s: the string to be parsed
//       variables: provided variables

//first, split s into sub terms as a list of strings
//for examples, "con 1 (con 2 Nil)" will be split into
//  ["con", "1", "con 2 Nil"]
List terms = splitTerm(s);

//now translate Strings to Terms
if (terms.size()>1){ //must be a function with parameters
  //create the function
  Function f = createTerm(terms[0], variables);
  //create the term objects for the parameters
  for (int j=1;j<terms.size();j++) {
    Term t = createTerm(terms[j], variables);
    f = f.addParameter(t);
  }
  return f;
}else if (terms.size()==1){
  //can be a value or a function without any parameter
  String x = terms[0];
  if (isValue(x))
    return createValue(x);
  if (x.startsWith("%")){ //isVariable
    return variables[getVariableIndex(x)];
  }
  //must be a function's name
  String className = x;
  return createFunction(className);
}
else throw new RuntimeException(``empty string'');
```

With the parsing algorithm, the creation of terms is a lot more elegant than using the native Java code:

```
  // cons 1 Nil
  Term list1 = Term.createTerm(``cons 1 Nil'');
  // demonstration of variables: cons 10 list1
```

```
Term list2 = Term.createTerm(``cons 10 %1'', list1);
// cons 10 (cons 1 Nil)
Term list3 = Term.createTerm(``cons 10 (cons 1 Nil)'');
```

3.3 Evaluation and Type Checking

The core of JFun is to evaluate each term, and reduce the terms into their simplest forms. Without considering lazy evaluations, the evaluation algorithm for a function can be described as follows. First, we check if enough parameters have been provided for the reduction. If so, we reduce all given parameters to their simplest forms, i.e. we recursively call the evaluation procedure on all parameters. Otherwise, we stop the evaluation process because it gets stuck by the definition of the function. Second, we transform the terms into the reduced format according to the rules defined in the function. For example, when we evaluate function `add`, we first check if there are at least two arguments provided. If not, we cannot continue evaluating the function. If so, we evaluate the two parameters x and y. Both of them are anticipated to be reduced into integers. Then, we calculate $x + y$ and create an `IntValue` object as the return value. Note that if more than two objects are provided, an exception will be raised because no parameters can be applied to any primitive types.

Type checking is done along with the evaluation process. That is, we do not have a static type checking mechanism before function executions. Instead, type checking is *dynamic* in JFun. In the `add` example above, if either one of the two arguments x and y does not end up with an integer, a type mismatched exception will be raised and the evaluation process will be terminated.

The above evaluation algorithm works well until we need lazy evaluation. Lazy evaluation is a unique feature of functional programming languages. It allows the existence of potentially infinite data structures. Therefore, we cannot evaluate all the parameters to their simplest forms before evaluating the function, because the parameter evaluations may never end.

The strategy we use to support lazy evaluation is two folds. First, in addition to having an `eval()` method of each term that reduces the term to the ultimate form, we also have a `lazyEval()` method so that the evaluation progress is only made one step forward. That is, as long as the evaluation makes some progress, `lazyEval()` will return the updated term. Hence, calling `eval()` on a term is equivalent to repeatedly calling `lazyEval()` until no further progress can be made. Second, we use *type-based pattern matching* on the parameters instead of blindly evaluating all parameters before reducing the current function. The type-based pattern matching rule dictates that as long as the variables needed for the function match the types extracted from the parameters, we can stop evaluating the parameters and start applying the function rules to the variables. Otherwise, we use lazy evaluation on the parameters with mismatched types, until either they match or they cannot be evaluated further.

For example, considering the function `add x y = x+y` again. If we annotate type `IntValue` with both `x` and `y`, then we should lazily evaluate the first two parameters until their types match `IntValue`. In this case, there is little difference from the implementation we described earlier without lazy evaluation consideration, because no further progress can be made on terms with primitive types. However, considering the following case:

```
repeat f a = cons a (repeat f (f a))
double x = x*2
second (cons a (cons b rest)) = b
```

We want to evaluate `second (repeat double 10)`. Simply evaluating the first parameter of the function `second` would incur an infinite recursion loop. Instead, in lazy evaluation, we will do a pattern matching between the wanted type `(cons a (cons b rest))` and the given parameter `repeat double 10`. Even we do not annotate any type to `a`, `b`, or `rest` (which means that they are allowed to match terms of any type), the types of the whole terms apparently do not match. Thus, a `lazyEval()` call on the term `repeat double 10` is executed, which returns

```
cons 10 (repeat double (double 10))
```

`lazyEval()` is continuously called on the returned term until the type completely matches the expected one. In this example, it stops when the parameter is reduced to

```
cons 10 (cons (double 10) (repeat double (double (double 10))))
```

Based on the pattern matching, `b` is extracted as the term `double 10`, which is further evaluated into the final result (20) based on the definition of `double`. The pseudo code of the lazy evaluation implementation in function `second` is showed in Listing 4.

Listing 4: pseudo code of lazy evaluation in function second

```
//evaluate the term to the simplist form
Term Function.eval() {
  Term res = this;
  // is it fully evaluated?
  while (!res.evaluated)
    res = res.lazyEval();
  return res;
}

//second (cons a(cons b rest)) = b
Term FJsecond.lazyEval(){
  //need at least one parameter to proceed
  Term tmp = checkParams(1);
  if (tmp!=null) return tmp;
  Function cons = parameters[0];
  while (!cons.getName().equals("cons")){
    cons = (Function)cons.lazyEval();
  }
  cons = cons.parameters[1];
  while (!cons.getName().equals("cons")){
    cons = (Function)cons.lazyEval();
  }
  Term b = cons.parameters[0];
  //attach the remaining unused parameters if any
} return attachRemainingParams(b,1);
```

3.4 Runtime Functions

Being able to do lazy evaluation in JFun allows us to implement all the functions provided in [1]. To add a new function, one needs to extend the class `Function`, and override the `eval()` with laziness consideration. However, for most of the functions with relatively simple implementation, the work of creating a new class is an overkill, and tedious as well. Besides, all functions must be programmed at compile time with explicit corresponding Java classes, which reduces flexibility.

To overcome these shortcomings, we provide the support of creating functions at runtime, i.e. *runtime functions*. The syntax of using runtime function is as follows:

```
RuntimeFunction::= Function [Arguments] = Implementation
Arguments::=Term [Arguments]|(Arguments)
Implementation::= Term [Implementation] | (Implementation)
```

```
Term::=Function | Variable
Variable::=$VariableName|$$VariableName
```

Here are some self-explanatory examples.

```
addRuntimeFunction("quadruple $$x=add (double $$x) (double $$x)");
addRuntimeFunction("map2 $f = reduce (dot cons $f) Nil");
addRuntimeFunction("second (cons $a (cons $b $rest))=$b");
```

Note the subtle differences between the variables with prefix of $ and $$. A variable with a prefix of $$ means that the expected type is one of the primitive types. On the other hand, a variable starting with $ can be matched to any type. In the `quadruple` function, since we are expecting to take an integer, `$$x` is used. The function of `second` only needs to return the second element of the list, where no type assumption is made. Therefore, `$b` is used instead.

Providing the type hint for the runtime function has two benefits. First, if the program is not well typed, the error can be observed in the early stage of evaluation. Second, it increases the efficiency of the program. In the `quadruple` example, if variable `$$x` is given as a very complex term, lazily plugging it in the implementation term would make the runtime system evaluate it twice.

Our current implementation of the runtime functions is still rudimentary. For example, no "if-then-else" or "where" clause is supported so that we cannot implement all the functions we need as runtime functions. Having a full-blown runtime function implementation is very similar to build a compiler and runtime system of the functional programming language.

4 Examples

Although it is not usually apparent, many Java developers encounters closures and higher order functions in their development practices. For example, many Java developers first encounter closures when they enclose a lexical unit of Java code within an anonymous inner class for execution. This enclosed unit of Java code is executed, on demand, by a *higher order function*. For example, the Java code in Listing 5 encloses a method call within an object of the type `java.lang.Runnable`.

Listing 5: closure example

```
Runnable worker = new Runnable(){
  public void run(){
    parseData();
  }
};
```

The method parseData is literally enclosed (hence the name “closure”) within the Runnable object instance, worker. It can be passed around between methods as if it were data and can be executed at any time by sending a message (called run) to the worker object.

Another example of the usage of closures and higher order functions in an object-oriented world is the Visitor pattern. Basically, the Visitor pattern presents a participant called the Visitor, whose instances are accepted by a composite object (or data structure) and applied to each of the constituent nodes of that data structure. The Visitor object essentially encloses the logic to process a node/element, using the accept (visitor) method of the data structure as the higher order function to apply the logic.

In order to illustrate more specific examples, we implemented the functions given as examples in [1]. The rest of this section summarizes the implementation details of some of those examples.

4.1 Simple List Processing Functions

Adding up the numbers in a list, multiplying the numbers in a list, and appending one list of numbers to another list of numbers are some of the functions that we have implemented. A `list` is basically defined by

```
listof X ::= nil | cons X (listof X)
```

where a `cons` represents a list whose first element is the X, which may stand for any type such as “integer”, and whose second and subsequent elements are the elements of the other list of Xs.

We can define a recursive function `sum` which will operate on a list and add the first element of the list to the sum of the others. Therefore, the definition of `sum` can be given as

```
sum nil = 0
sum (cons num list) = num + sum list
```

The key part here to notice is that only result of the nil case and the addition operator (+) are specific to the computing a sum. Therefore, this definition can be parameterized and the `reduce` function, which will be used as a template for applying a function `f` to some `x`, can be derived.

```
(reduce f x) nil = x
(reduce f x) (cons a l) = f a ((reduce f x) l)
```

Hence, `sum` can be redifined easily in terms of `reduce` function pattern as well as creating new functions, such as `product`.

```
sum = reduce add 0
product = reduce multiply 1
```

Java implementation of `sum` function, renamed as `FJsum`, is provided in Listing 6.

Listing 6: Java implementation of `sum` function

```
public class FJsum extends Function {
  public FJsum() {}
  public FJsum(Term list) { super(list); }
  public Term eval( boolean lazy ) {
    Term tmp = evalParams(1, lazy);
    if ( tmp != null ) return tmp;
    if ( parameters.size() > 1 )
      throw new RuntimeException(``sum: more than 1 arg'');
    FJreduce r =
    new FJreduce(new FJadd(), new IntValue(0),
                 (Term)(parameters.get(0)));
    return r.eval();
  }
}
```

4.2 Newton-Raphson Square Roots

Implementation of this method is a very good way of illustrating the power of lazy evaluation. Newton-Raphson algorithm computes the square root of a number N by starting from an initial approximation a0 and improving this approximation using the rule

```
a(n+1) = (a(n) + N/a(n)) / 2
```

If the approximations converge to some limit `a`, then

```
a = (a + N/a) / 2
2a = a + N/a
```

```
a = N/a
a*a = N
a = squareroot(N)
```

Listing 7 shows the Java implementation of this function, renamed as `FJsqrt`.

Listing 7: Java implementation of `sqrt` function

```java
public class FJsqrt extends Function {
  public Term eval( boolean lazy ) {
    Term tmp = evalParams( 3, lazy );
    if( tmp != null ) return tmp;
    Term res =
         createTerm(``within %2 (repeat (next %3) %1)'',
                    (Term)parameters.get(0),
                    (Term)parameters.get(1),
                    (Term)parameters.get(2));
    return attachRemainingParams(res,3,lazy);
  }
}
```

In this implementation, an example of creating a new `Term` via parsing a string can be seen.

4.3 Alpha-Beta Heuristic

Alpha-Beta heuristic is an algorithm that is used for estimating how good a position a game-player is in. This algorithm is mainly used in board game programs, such as chess, backgammon, and tic-tac-toe, and works by looking ahead to see how the game might develop.

The first stage is the construction of a gametree, which is essentially a tree of game positions. An example gametree for tic-tac-toe game is illustrated in Figure 2. If we also have a function, call it `moves`, that takes a position and generates a tree of possible positions that can be reached from that position in one move, we can easily define a new function for generating a gametree as

```
gametree p = reptree moves p
```

where `reptree` applies the given function on every node of the tree. After generating the gametree, we can use the function composition notation (.) and write an evaluation function easily by using the concept of lazy evaluation and the basic building blocks that have already been implemented as

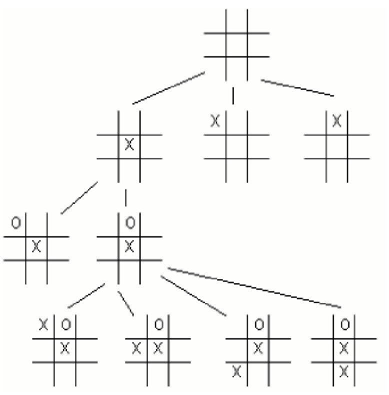

Figure 2: an example of a gametree for tic-tac-toe

```
evaluate = max.maximize'.maptree static.prune n.gametree
```

where `prune n` prunes the tree at a specific depth in order to prevent it from growing forever, `static` evaluates the static board position and assigns a number to the position based on its quality, `maptree` applies the static function to all available nodes in the tree, and finally `max.maximize'` sets the minimax value calculated on that tree to the current position.

Although those algorithms seem to be difficult, the concepts of modularization and code re-use, which are the essence of functional programming, make it very easy to implement.

5 Conclusions

In this paper, we provided a new design style for implementing functional programming concepts such as modularity and lazy evaluation in an object-oriented programming environment which has been chosen to be Java. It has long been clear that modularity is the key to productive and successful programming on any platform. The problem for Java developers is that modular programming entails more than just decomposing a problem into parts; it also involves being able to glue small scale solutions together into an effective whole. Since this type of development is inherent to the functional programming paradigm, it seems natural to use functional programming techniques when developing modular code on the Java platform. As it may easily be seen, closures and higher order functions are not completely unfamiliar programming concepts for most Java developers, and they can be effectively combined to create a number of very handy modular solutions.

In order to make the concept easier to understand, we provided some examples throughout the paper, which are originally taken from Hughes' paper titled "Why Functional Programming Matters" [1], and implemented in Java.

References

John Hughes. "Why Functional Programming Matters". The Computer Journal, 2(32):98-107, April 1989.

Jeroen Fokker. "Functional Programming". Department of Computer Science, Utrecht University, 1995.

Haskell - A Purely Functional Language. "http://www.haskell.org/".

Java Programming Language. "http://www.java.sun.com/".

Gilad Bracha. "Generics in the Java Programming Language", 2004.