Java/MP Project Report

Roshan Naik,
Department of Computer Science
Oregon State University
August 2001

Major Advisor: **Timothy Budd**

Java/MP : A Multiparadigm Programming Language

by

Roshan Naik

A project report
submitted to

Computer Science Department,
Oregon State University

In partial fulfillment of
the requirements for the
degree of

Master of Science

Completed August 14, 2001
Commencement June 2002

Abstract

In this report we present Java/MP, a multiparadigm language designed as an extension to Java. Java/MP is an upward compatible superset of Java and incorporates the object oriented, functional and logical paradigms. Java/MP programs are compatible with the standard Java virtual machine. Many of the ideas in Java/MP have been borrowed from an earlier multiparadigm language Leda which is based on Pascal. One of the objectives in the design of Java/MP was to add as few features as possible and make the extensions feel natural and evolutionary. A prototype compiler was developed as part of this project. Java/MP is currently implemented as a preprocessor that translates Java/MP code into pure Java.

This report describes the features new to Java/MP and also the translation techniques used by the prototype compiler. We will also take a look at some sample programs written in Java/MP.

Acknowledgements

A fair amount of work went into this project and I would like to acknowledge all the people who contributed directly or indirectly in many ways. I am grateful to my major professor Dr. Timothy Budd for many things. For teaching me compiler writing, introducing me to the exciting world of multiparadigm programming, offering all those wonderful courses, guiding me through the project and giving me this opportunity to work with him. Many thanks to Dr Margaret Burnett for introducing me to the different programming paradigms. Thanks to Dr. Bose Bella and Dr. Martin Erwig for serving on my graduate committee. A large amount of help came indirectly from Dr. Mark Abbott (Department of Oceanography), under whom I served as a research assistant in the Department of Oceanography. Life would have been substantially harder for me without his help. Always grateful to my parents for mentoring me the way they have and for their invaluable love and support.

Must thank the people at GNU for making flex and bison freely available and also the people at Borland for C++ Builder without which the development process would have been so much harder. Also thanks to Dmitri Bronnikov for making his Java grammar for lex and yacc freely available.

Table of Contents

Chapter 1. Introduction

1.1 Background

The American Heritage Dictionary defines the word paradigm as follows:

A set of assumptions, concepts, values and practices that constitutes a way of viewing reality for the community that shares them, especially in an intellectual discipline.

So paradigms are fundamentally characterized by the way a community uses them to view reality. Paradigms are perhaps best illustrated in the religions. Every religion subscribes to a set of ideas and the religious community shares them. Some religions may have common set (small or large) of ideas, in which case they could be categorized into a common paradigm.

Programming languages can also be divided into different paradigms. These paradigms however are about problem solving. Programming languages are used to instruct the computer and the computer is a device for doing computation or problem solving. This basically leaves a programming language responsible for providing effective problem solving tools. The paradigm or the school of thought to which a programming language belongs directly influences the way in which problems are tackled using the language. That is, the technique invented to solve a given problem in one language can be strikingly different than the technique that would be invented in another language subscribing to a different paradigm.

Let us examine some of the major paradigms popular among programming languages today.

1.1.1 Imperative Paradigm

This paradigm is founded on the Turing / von Neumann ideas of computation. Simply put, computation is performed by a executing a series of instructions that read data from some memory and write intermediate and final results back into memory. The instruction sequence continually makes incremental changes to memory. Programming languages such C, Pascal and Fortran are the well known examples of imperative paradigm. Plankalkül is believed to be the oldest (imperative) high level language. High level languages simplify the task of dealing with memory by assigning names to portions of memory. A named area in memory is called a variable. To simplify creating a sequence of commands that is executed, several control structures are provided in these languages. For example,

1. Loops for repeated execution of a set of instructions. For example, the C language provides `for` and `while` constructs for looping.
2. Test instructions that allow selective execution of instructions. For example, the `If then` statement in Pascal.
3. Grouping set of instructions into a unit called a function. Like variables, functions are referred to by name.
4. Branch instructions to jump from one point in the instruction sequence to another. For example, the `goto` statement and the function call mechanism in C.

1.1.2 Functional Paradigm

The functional paradigm is based upon a branch of mathematics called lambda calculus that has its roots in Alonzo Church's work on recursive functions. In this paradigm there are only constant values and functions. There is no notion of incremental changes being made to portions of memory to reflect computation results. Functions are similar to algebraic equations involving variables. The process of substituting the variables with values is referred to as "applying". Just as in mathematical equations, once the variables are assigned values, they do not undergo change during the process of evaluating the function. Results are obtained when a function is applied with some values and each time the same values are applied to a function the same result is obtained[1]. A function is the fundamental building block of the functional paradigm. Functional languages allow functions themselves to be treated as values. Consequently a function itself can be applied to another function and also the result of applying a function can be a function. This property is commonly referred to as **higher order functions**. Functional languages use the following two important ideas to build programs with functions:

Recursion: The idea that a function can be defined in terms of itself. For example the factorial of an integer `n` is the factorial of `n-1` multiplied by `n`. Thus factorial can defined as follows,

```
factorial(0) = 1
factorial(n) = factorial(n-1) * n
```

Composition: The idea that a function can be defined in terms of other functions. Thus if we have a function `square` that evaluates the square of an integer `n`, we can use it to write a function that computes the cube as follows

```
cube(n) = n * square(n)
```

John McCarthy's LISP is perhaps the most well known functional language. Others include ML, Scheme (a LISP dialect) and the more recent Haskell [Haskell, 1999].

1.1.3 Object Oriented Paradigm

The most fashionable paradigm today is undoubtedly the Object Oriented paradigm (OO). As the name suggests, the concept of an "object" is the most fundamental idea. The world can most naturally be viewed as interactions among objects. Complex systems can be decomposed into constituent entities and each entity viewed as an object that exhibits some behavior. Objects interact with each other by passing messages and computation is performed by objects in response to the messages received. An object consists primarily of data and behavior. Objects that exhibit the same behavior belong to the same "class". A class simply describes the nature of data contained and behavior exhibited by a group of similar objects. Classes can be categorized in a hierarchical manner to group related classes under a parent class. Such child classes inherit data and behavior from parent classes. This inheritance mechanism proves to be a very useful technique for code reuse. Some languages such as C++, Python, and Eiffel [Budd, 1995] allow a class to inherit from more than one class. This feature is called **multiple inheritance**.

Most object oriented languages such as C++, Java and Delphi-Pascal, provide the object oriented paradigm as an extension to the imperative paradigm. Other languages such as Mondrian (Haskell dialect) and Common Lisp (Lisp dialect) extend the functional paradigm with the object oriented features.

[1] This is an important property of functional languages and is called referential transparency.

1.1.4 Logic Paradigm

David Warren[2] notes,

"The key idea behind logic programming is that computation can be expressed as controlled deduction from declarative statements".

The programming style in this paradigm is declarative as opposed to the procedural nature of the imperative and object oriented paradigms. Explicit sequential command sequences or actions is absent in the logic paradigm.

Logic programming has its roots in predicate calculus. Problem solving is done by inferencing results using what is already known. This mechanism is often similar to how the human mind comes to conclusions or the manner in which proofs are solved in mathematics. The basic idea is that the system starts off with some knowledge about the problem domain. This knowledge is represented in two ways, facts and rules. A fact represents a well known piece of information that is not dependent on any other information. In other words, it's a self evident truth. Logic programs have a collection of such facts that form the basis (a knowledge repository) for problem solving. Rules dictate the relationships between different pieces of information and provide the rules with which new inferences can be deduced. Rules can refer to facts, other rules and itself.

This technique is similar to theorem proving. A theorem is really an inference and in order to deduce a new inference we often make use of other known facts and rules. Finally a problem instance that needs to be solved is proposed as a query to a system loaded with facts and rules form the problem domain. This causes the logic system to begin a search for an application of the rules and facts that will provide a solution to the query. Basically this is an exhaustive search for a solution governed by the defined rules. Some paths taken during the search lead to no useful conclusions in which case the system backtracks and tries other options if any. Finally when all its options are exhausted and if it hasn't found a solution it gives up. This means that the problem instance formulated as the query is not solvable using only the information currently known to the system. Otherwise, the solutions found are returned as results for the query.

Prolog is the oldest and best known logic programming language. Gödel [Gödel, 1994] is a more recent logic programming language designed as a successor to Prolog.

1.1.5 Generic programming Paradigm

This paradigm is usually implemented as an extension to the imperative and object oriented paradigms. The goal of generic programming is to maximize reuse of algorithms over types. For example, a quicksort algorithm implemented for a string will typically be implemented in much the same way for another type such as a int or double. The only significant difference in the implementations would be the logic used to compare values. Integers use a different comparison technique than strings. Code reuse can be maximized if the sorting algorithm is shared for all types and only the comparison functions are implemented separately for each type. Generic programming allows such code sharing for common algorithms by not fixing upon the types in the implementation for a generic function. So a generic sort function is implemented over an arbitrary type. This idea is extended to classes in object oriented languages. Thus classes can specified to contain members of arbitrary types.

Generic functions and classes often require their types to support certain operations depending upon what the types are used for. For example, in the case of sorting, it is required that the type should allow comparison

[2] David H. D. Warren is credited for designing the Warren Abstract Machine that forms the basis for implementing Prolog and other logic programming languages.

operations. Some languages such as Eiffel [Eiffel, 1992], Leda [Budd, 1995] and Pizza [Odersky, 1997] allow such requirements to be specified explicitly in the declaration of the class or function. This is often referred to as **bounded** or **constrained genericity**. C++ which also allows generic programming does not provide any direct support currently to constrain generics.

1.1.6 Multiparadigm

There is a plethora of paradigms that exist on which programming languages are based. Some other paradigms include the visual, spreadsheet, rule based, access oriented, subject oriented, constraint etc. Languages that subscribe to more than one paradigm are called multiparadigm languages. Smooth and seamless integration of the paradigms then becomes an important issue for these languages. Another important issue is the degree to which each of the paradigms is supported. The prime motivation for research in multiparadigm languages stems from the fact that not all problems are best solved within a single paradigm. Problems that can be solved very easily in one paradigm can prove to be a sizeable undertaking in another. For example it is widely accepted that logic programming languages prove to be a rather useful implementation tool for expert systems but lack good support for numeric computation which is a forte of imperative languages. Object oriented languages on the other hand are the most natural fit for creating simulations of real world systems.

Some multiparadigm research tends to provide a greater emphasis on providing maximum support paradigm to each individual paradigm but compromise on smooth integration between them. Others tend to provide seamless integration among the paradigms but provide a very high degree of support to one paradigm and heavily compromise on others. [Diomidis,1994] discusses some ideas on implementing a multiparadigm system. They introduce an abstraction named *call gate*. The call gate provides a communication channel for the interaction between the different paradigms. Every paradigm will provide call gates to allow other paradigms to interact with it. This technique makes the multiparadigm system look like a conglomeration of mutually independent paradigms. Even though a high degree of support can be given to any particular paradigm, mixing paradigms seamlessly turns out to be elusive.

Multiparadigm languages differ vastly in the paradigms they support. The Oz programming language, for instance, supports logic, object-oriented, constraint, and concurrency programming. Pizza supports functional, generic and object oriented paradigms. Leda supports the functional, object oriented, generic, and logic paradigms.

Java/MP follows in the footsteps of Leda and offers the functional, object oriented, and logic paradigms but lacks the generic paradigm[3]. Java/MP's approach to providing a multiparadigm system is, like in Leda, by extending the object oriented paradigm with the functional and logic paradigms.

1.2 Motivation

The object oriented paradigm has become the dominant programming paradigm today. Its has become a part of academic programs in many universities and familiarity with its basic concepts is widespread. Java too has grown to be a popular object oriented language in academia and industry. Its simple object model provides a short learning curve for beginners, while its “write once, run anywhere” promise has given a tremendous impetus for professionals seeking to create platform independent applications. The ability to create applications in Java that run inside a web browser as a front end to a web application or as a backend in a web

[3] Sun has recently previewed a version of Java with support for generics but has not yet included generics in the standard. Thus adding generics into Java/MP at this point in time could create intolerable incompatibilities with future Java versions.

server is another big advantage for Java programmers wishing to leverage their skills. Besides having a strong user acceptance it also seems to be growing in popularity and evolving to meet programmer requirements in various ways.

We feel that Java suffers from one problem. It confines programmers rather strictly within the object oriented world. A good programming language should allow a fair amount of flexibility to the programmer for abstracting and solving problems in the language. The language should let the programmer choose an appropriate solution rather than force one upon him. Most other object oriented programming languages typically include a few features from other paradigms and thus provide a greater degree of freedom. For example, Ruby [Ruby,2000] is primarily an object oriented language, but allows a function to be passed as an argument to another function in a somewhat restricted manner[4]. Python, also primarily an object oriented language, provides support for higher order functions by treating all functions as objects and providing facilities for declaring lambda[5] functions. Eiffel and C++ support the generic paradigm.

The multiparadigm language Leda has not only demonstrated that a multiparadigm system concocted using the logic, functional and object oriented paradigms can be implemented, but also showed that these paradigms can be integrated seamlessly. As Timothy Budd notes:

"… a synergy quickly develops between the different paradigms, and a great deal of power derives from combining code written in two or mores styles in a single function."

Building a multiparadigm language that is defined as a strict superset of Java and translating it into Java instantly brings in advantages such as:

1) The language can make use of the vast library support that already exists for Java.
2) The Java runtime environment which includes a well designed security system, automatic garbage collection and other such features can be leveraged without any extra effort.
3) Interoperability with existing Java programs is ensured.
4) Features such as platform independence and deploying programs on the Internet are obtained for free.

It is not surprising that the most complex features of a language are the ones that are used least. Thus, when extending a language it is important to keep extensions simple without sacrificing on the ideas that motivated the extensions in the first place. Also retaining the original feel of the language is key to making the extensions feel natural and evolutionary to existing programmers.

[4] Only one function (more precisely a "block of code") can be passed as argument to another function.
[5] Refer to section 2.1 for a discussion on Lambda functions

Chapter 2. Java/MP Language Overview

Java/MP is a multiparadigm programming language designed as an extension to the Java programming language. The language has been conceived and designed by Timothy Budd. It is really a new version of the earlier multiparadigm language Leda. Java/MP is an upward compatible superset of Java. In addition to the object oriented style present in Java, it introduces functional and logic programming styles. One of the interesting aspects of Java/MP is that it introduces very few syntactic additions to Java and also maintains the original feel of the Java language. The language is compatible with the Java VM, and implemented as a source-to-source transformation. So the output generated by the prototype compiler is pure Java code, which is in turn fed to a Java compiler to produce byte code. This chapter provides a brief description of the important features new to Java/MP.

2.1 Functional Style Programming

Higher order functions are an important feature of the functional paradigm. Functions can be treated as values. Thus a function can be passed as an argument to another function or be returned as a result. Support for higher order functions requires two important features:

1) Variables of function type: The type of a function is determined by the formal arguments (types and arity[6]) and the return type of the function. Consider functions `square`, `cube`, `squareRoot` and `IntegerToInt`;

```
int square(int value)
{
   return value * value;
}

int cube(int value)
{
   return value * value * value;
}

int product( int value1, int value2 )
{
   return value1 *  value2 ;
}

int IntegerToInt( Integer value)
{
   return value.intValue();
}
```

Functions `square` and `cube` have the same type since they have the same arity and argument types. The type of function `product` differs from the former two functions due to the difference in arity. Function `IntegerToInt` has the same arity as square and cube but differs in its argument type. Thus function `IntegerToInt`'s type is different from the types of `product`, `square` and `cube`. In Java/MP a function can be implemented in the global scope, as a member function (static or a non static) of a class or as a lambda function. But the function's type is independent of how the function is implemented.

[6] The number of arguments taken by a function is known as arity.

In order to treat functions as values we need variables that can store functions. The following declares a variable named x, which can be used to store a function that takes a single `boolean` argument and returns an `int`. Notice use of square brackets to denote the function type and also the lack of parameter names.

```
        [int(boolean)] x;
```

`[int(boolean)]` denotes a function that returns an `int` and takes a single `boolean` argument. Thus `x` can be used to store a function of this type. If a function `bar` is declared as follows,

```
        int bar(boolean x)
        {
           //…
        }
```

then we could assign the method `bar` to the variable `x`

```
        x = bar;
```

The function `bar` can be a global function, non-static member function or a static member function.

2) Functions as expressions: Functions when used as expressions are called **lambda** (or anonymous or nameless) functions. Lambda functions do not have names and can be used as expressions. Following code shows the syntax for lambda function,

```
        [int(boolean b)]
        {
           if(b==true)
                 return 0;
           return 1;
        }
```

Since the parameters will be required within the function, they obviously need to be named. This function can be assigned to the variable `x` as follows.

```
        x = [int(boolean b)]
            {
                 if(b==true)
                     return 0;
                 return 1;
            };
```

Lambda functions can be passed as arguments or returned as values from other functions. Consider a function `foo`, that takes a single function type argument called `func`, and returns the value obtained by invoking `func`

```
        int foo([int(int, int)] func)
        {
           return func(2, 3);
        }
```

`foo` can be invoked as follows

```
        foo( [int(int x, int y)]
              {
                return x+y;  // or x*y or whatever !
              }
```

```
      );
```

Or

```
   foo(z); // assuming z has been initialized with a function
```

Scoping rules allow anonymous functions to refer to local and instance variables from the surrounding scopes. Since the life of a lambda function can possibly exceed the life of its surrounding function, the language implementation takes care of storing context information appropriately. One artifact of the current implementation of Java/MP is that the changes made to local variables will not persist beyond the scope of the lambda function making these changes. But changes made to instance variables will persist beyond the scope of the lambda function. This is different from the Leda implementation.

2.2 Logic Programming

Pass by name, **operator overloading**, **global functions** and **relations** are new features in Java/MP. Support for logic programming is provided using a combination of these features along with higher order functions.

1. **Pass by name:** A function can be defined to accept its parameters by name as follows

```
   int foo( int @ x , int y)
   {
      y = x+1;
      return x + y;
   }
```

The `@` symbol in the declaration of parameter `x` is used indicate that it will be passed by name. The value of a by name parameter is re-evaluated every time it is needed. If `foo` is invoked as follows

```
   int a=4 , b= 5;
   foo(b*a, 25);
```

The value of `b*a` will be evaluated twice within `foo`. Once while evaluating the expression `x+1` and a second time for `x+y`. Since the variables `a` and `b` are not ordinarily available within the body of `foo` the implementation will create a *thunk* each time an argument needs to be passed by name. Conceptually a thunk is just a nameless argumentless function that packages the expression. The thunk will need to hold on to all necessary context information necessary (in this case values of variables `a` and `b`) and also allow the expression to be re-evaluated whenever needed. Since context may be stored in the thunk, it is implemented as a class. Sections 3.6 and 3.7 discuss the details involved in creating thunks.

One limitation of the pass by name mechanism is that a byname variable can be assigned a value only if it has been initialized with a variable. If an assignment is done to a by name variable that is initialized with an expression, an error will occur at run time. In the above call to `foo` we have initialized `x` with the expression `a*b`. Thus any assignment to `x` within `foo` will be an error. But if we invoke `foo` as

```
   foo(b,25);
```

an assignment to `x` within `foo` will be safe.

Pass by name is an important tool that allows us to smoothly integrate the logic paradigm with the object oriented and functional paradigms. It gives us the ability to have information flow in and out of a function through a parameter list, similar to that of a pass by reference mechanism in C++. Java implements pass by value and a variation of pass by reference, which is in fact, a pass by value of reference. This means that, when passing an object by reference, a copy of the reference to the object is actually passed. Consequently, when a new value is assigned to this reference within the callee, the change in value is not reflected back in the caller. Java's lack of supoort for a true pass by reference mechanism can be noted by the fact that one cannot implement a *swap* function in Java.

Consider functions `foo` and `bar` declared as follows

```
boolean foo(int a, int @ b)
{
   b = 4;
   return true;
}

boolean bar(int @ c , int d)
{
   if(c==4)
      return true;
   else
      return false;
}
```

If we invoke them as follows,

```
int x = 1, y = 2, z = 3;
if( foo(x, y) && bar(y, z) )
{
   // …
}
```

the implementation will create a thunk for `y` while applying `foo()` and then reuse the thunk while applying it to `bar()`. So inside `bar`, the value of `c` will indeed be `4`.

2. **Operator overloading and global functions:** Global functions are conceptually simple and similar to that found in C/C++. Global variables however are not allowed in Java/MP. Operators overloading in Java/MP is slightly different than in C++. In Java/MP the meanings of the operators defined by Java cannot be changed. So addition operator + cannot be overloaded on `ints`. Assignment operators = and `<-` (relational assigment) cannot be overloaded. The other important difference being that overloading is done using names assigned to operators. Operator overloading is always done in Java/MP in the global scope. So the addition operator can be overloaded on a user defined type `complex` as follows

```
complex plus( complex x, complex y)
{
  complex ret = new complex();
  ret.real = x.real + y.real;
  ret.img = x.img + y.img;
  return  ret;
}
```

3. **Relations:** Java/MP introduces the notion of a *relation* which can be considered an extension of boolean. Relations are useful for declarative style programming similar to that of Prolog (horn clauses). Relations are really functions that return a boolean value. In addition to returning a boolean, they also take another relation as parameter. Thus it has an interesting recursive definition. Java/MP implementation defines the relation as a Java interface

```
interface relation
{
    public boolean apply(relation continuation);
}
```

The Java/MP compiler performs automatic conversions from boolean to relation and back, whenever a boolean is used in place of a relation or vice versa.

Among the most novel features of Java/MP (as in Leda) is the complete support for backtracking and unification provided in a few simple functions as part of its standard library. Java/MP introduces the relational assignment operator `<-` (also found in Leda). It is provided a default implementation in the standard library and cannot be overloaded. This operator essentially performs a reversible assignment. That is, the assignment is undone if a certain condition is not true. Operators `&&` and `||` are provided default implementations for relation arguments. The `||` operator is what enables the backtracking mechanism to pursue alternatives. Implementations of each of these operators is done in remarkably few lines of code and based on chapter 22 of [BUDD, 1995]

2.3 Logic programming examples in Java/MP

The following example provides a simple illustration of back tracking

```
int a =3;
if ( ( (a<-5) || (a<-2) ) && ( a < 3 ) )
    System.out.println(a);
```

The print statement will be executed when the `if` condition is satisfied. First `a` will be temporarily assigned the value `5` and then the condition `a<3` is given to the `<-` arrow operator. Since the condition fails the value of `a` is restored to `3` and the `||` operator will force the execution of the other alternative, `a<-2.` This will again temporarily assign `2` to `a` and check if `a<3`. This time the condition holds and the print statement will be executed. Finally the value of `a` is restored back to the initial value `3` and execution will proceed beyond the `if` statement.

The `if` statement will look for the first solution only. Sometimes we are interested in all the solutions. In such cases we can use the `while` loop instead. Consider a similar example,

```
int a = 3;
while( ( (a<-2) || (a<-4) || (a<-6) ) && (a>3) )
   System.out.println(a);
```

When `a` is bound to `2` the condition `a>3` will fail, forcing the backtracking mechanism to try other alternatives. The next binding of `a` to `4` will cause the condition `a>3` to hold `true`. The body of the while loop will be executed. At this point the implementation lures the `||` operator into believing that the condition did not really hold true, effectively forcing the exploration of other alternatives. This continues till all alternatives are exhausted. Finally, after the execution of while loop, the value of `a` is restored back to `3` and the execution proceeds.

The use of `if` and `while` constructs to iterate over solutions is a good example of how the imperative paradigm blends smoothly into the logic paradigm.

Also programmer defined relations can be used to construct more interesting examples.

```
relation child( String @ progeny, String @ mother)
{
   return ( eq(progeny,"William") && eq(mother,"Diana") )
          || ( eq(progeny,"Henry") && eq(mother,"Diana") )
          ;
}

relation mother(String @ ch, String @ mom)
{
    return child(ch,mom);
}
```

The function `eq()` is part of the standard library and performs unification. If its first argument is `null` it assigns the second argument to the first, otherwise it will test if the first argument is equal to the second argument. The `child` relation (actually a function that returns a relation) provides the database about mother and child relationships. The `mother` relation will make use the child relation in order to determine if `ch` is in fact child of `mom`. So we can test for mother and child relationship as follows

```
   if( mother(“William”,”Diana”) )
         System.out.println(“Diana is William’s mother”);
```

Notice that the mother and child relations take all their parameters by name. We noted previously that by name parameters enable bi-directional flow of information (in and out of functions). Since mother and child relation use by name parameters, we can reuse the same code to also enumerate all solutions to queries such as “Who are the children of Diana?” as follows,

```
String child=null;
while( mother(child,”Diana”) )
{
   System.out.print(child);
   System.out.println (“ is a child of Diana”);
}
```

Similarly, other queries such as “Who is the mother of Henry?” can be constructed easily.

Chapter 3. Java/MP to Java Translation

This chapter describes how Java/MP features are translated into Java by the prototype compiler.

3.1 Functions as values

As noted in section 2.1, higher order functions are an important feature of the functional paradigm. A function that takes a function as parameter or returns a function as result is called a higher order function. Let us consider the issues involved in treating functions as values.

3.1.1 Types For Functions

In programming languages, values are always associated with types. The first step in treating functions as values is to develop a technique for assigning types to functions. The type of a function is determined by the formal arguments (types and arity[7]) and the return type of the function. Consider functions `square`, `cube`, `squareRoot` and `IntegerToInt`:

```
int square(int value) // type: [int(int)]
{
   return value * value;
}

int cube(int value)   // type: [int(int)]
{
   return value * value * value;
}

int product(int value1, int value2 ) // type: [int(int,int)]
{
   return value1 * value2 ;
}

int IntegerToInt(Integer value) // type: [int(Integer)]
{
   return value.intValue();
}
```

Functions `square` and `cube` have the same type since they have the same arity and argument types. The type for function `product` differs from the former two functions due to the difference in arity. Function `IntegerToInt` has the same arity as `square` and `cube` but differs in its argument type.

For assigning types to functions we do not have many choices since Java recognizes only primitive types, classes and interfaces for specifying types. We choose to use an interface for this purpose. We create a unique interface for every function type[8]. The interface name is obtained by mangling the return and argument types. Thus, a function such as:

[7] The number of arguments taken by a function is known as arity.
[8] This is similar to the technique used in Pizza. Pizza uses an abstract class with an abstract apply method to represent each function type. The naming scheme used in Pizza for these classes is not clear.

```
int foo ( double[] x, char y )
{
   //…statements
}
```

will result in the following interface:

```
interface int_function_double_array_char_
{
   int apply( double[] p1, char p2 );
}
```

Section 3.5 describes the scheme used for generating interface names from function types. Interfaces are generated only once for each function type found in source code. The `apply` method is used to execute the function. If the function `foo` was declared in the global scope, it is converted into a class that implements the above interface.

```
class function_foo implements int_function_double_array_char_
{
    public static int foo(double[] x, char y)
    {
        //….statements
    }
    public int apply(double[] x, char y)
    {
        return foo(x,y);
    }
}
```

The first function will simply have the same name and implementation as the original global function. The second function is the implementation of the parent interface and useful in deferring the resolution of function calls to runtime. The reason for the two methods is to avoid unnecessary creation of an object when invoking a known function. Thus,

```
foo(new double[2], 'a');
```

can be translated into:

```
function_foo.foo(new double[2], 'a');
```

The static function is only useful in cases when the function to be called can be resolved at compile time. When variables of function type are used to invoke a function, the function to be invoked needs to be resolved at runtime. In this case the non-static `apply` function is used. Thus if `baz` is function type variable, then the following code:

```
baz();
```

is translated into:

```
baz.apply();
```

The next section demonstrates how the apply function is used for variables of function type.

3.1.2 Variables Of Function Type

Variables of function type are very useful in treating functions as values. They can be initialized with a function, assigned to other function type variables, passed as arguments or returned as results from functions. Like functions, function type variables can be invoked with arguments. This is an important difference between function type variables and regular variables. Consider `bar` declared as a variable of function type:

```
[int(double[], char)] bar;
```

The translation simply declares a variable of the interface corresponding to the function type:

```
int_function_double_array_char_  bar ;
```

When `bar` is assigned a named function (a global function or static member function or non static member function),

```
bar = foo;
```

it is translated it into:

```
bar = new int_function_double_array_char_ ()
          {
              public int apply(double[] x, char y)
              {
                   return function_foo.foo(x, y);
              }
          };
```

A new anonymous class instance, implementing the same interface, is assigned to `bar`. The implementation for the `apply` method here simply forwards the call to the appropriate function. So when `bar` is invoked as follows,

```
bar(new double[2], 'a');
```

we use `apply()` to resolve the call to function `foo` at runtime.

```
bar.apply(new double[2], 'a');
```

3.1.3 Context Saving

One problem in treating functions as values is that scoping rules allow functions to refer to variables from surrounding scopes. All variables from outer scopes that can be referenced by a function are collectively referred to as the function's **context**. When using functions as arguments or return values, we need to store the context information and make it available to the function when needed. Chapter 18 of the Leda book describes a technique that can be used by the run-time system to support context saving. Since modifying the Java runtime is not an option for Java/MP (for interoperability with existing Java code), we make use of classes to store context information. Section 3.2 below discuses this technique in detail.

It is important to note the difference between treating functions as values and treating them simply as pointers (memory addresses) as in C and C++. In C and C++, when a function is passed as a pointer to another function, there is no context that is carried along with the function pointer. This does not prove to be a problem in C since it allows functions to defined only in the global scope and variables from the global scope are accessible throughout the program. In C++, pointers to member functions can be used only if the member function is declared `static`. Declaring a member function as `static`, restricts it from referencing any instance variables belonging to its class. C++ however provides a technique called function objects to allow functions to be passed as arguments.

3.2 Functions as expressions

In a system where functions can treated as values, it is often convenient to treat functions as expressions. It simplifies the task of creating simple functions that will only be treated as values and never be invoked directly by name. If the function will never be referred to by name, it seems wasteful to assign a name to the function. Such a function can very well be specified "inline" where needed. This feature is supported to varying degrees in many object oriented languages such as Python, Smalltalk, Ruby and Eiffel.

To see how this can be useful, consider the following Java/MP code:

```
myList.onEach(
                [void(int x)]
                {
                   System.out.print(x);
                   System.out.print(',');
                }
              );
```

Here we see a single argument function being passed as argument to `myList.onEach()`. Let us assume that `myList` represents a list of integers and the `onEach` function will simply invoke the function passed to it on ever element in the list. The above code will then print each element followed by a comma. As we can see, the argument function has not been provided with a name. Such functions are called lambda functions (or anonymous or nameless functions).

The translation technique shown here for lambda functions is a combination of the techniques discussed in [Budd, 2000] and [Odersky, 1997]. The treatment given to lambda functions varies slightly depending upon characteristics of their surrounding scope. It is important to note that scoping rules allow lambda functions to refer to local variables declared in the function (named or anonymous) surrounding them. In addition they can also refer to instance variables declared in the class. There are three basic steps involved in the translation of a lambda function.

1) Create an interface from the lambda function's type, if the same interface has not already been created earlier.
2) Create a public member function called `lambda#` (# being an arbitrary number) inside the class where the lambda function is found. The body of this function is essentially identical to the body of the lambda expression. In addition to the parameters specified as part of the lambda function declaration, `lambda#` will accept context information (related to local variables from outer scopes) as parameters. Instance variables will be accessible to `lambda#` directly since it is a member function of the enclosing class.
3) A class called `context#`, implementing the interface from step 1, is created to hold the values of all locals required by `lambda#`. The implementation of the `apply` method in `context#` will accept the same parameters as the lambda function found in source. This function will simply forward calls to `lambda#` and pass on the context information to `lambda#` as arguments in addition to the standard arguments that it receives.
4) Create and use an instance of `context#` in place of the lambda function found in the Java/MP source. When instantiating `context#`, all locals that need to be saved in `context#` will be passed as arguments to its constructor.

There is a slight variation in the translation depending upon where the lambda function is found. If the lambda expression is found anywhere inside of a static or global function (named), then `lambda#` is declared as static. If the enclosing named function is neither static nor global, the `context#` class is declared as an inner class within the class surrounding the lambda function, else the class is placed in the global scope. When not enclosed in a static or global function, `context#` will also be initialized with a reference of the object within which `lambda#` is implemented. This allows `context#` to forward calls to `lambda#` belonging to the correct instance at run time. On the other hand if enclosing function is static or global, `context#` simply forwards calls using the name of the class inside which `lambda#` belongs.

Consider a lambda function being used within a global function `foo`.

```
int foo (double[] x, char y)
{
      [int (double, char)] bar = null;
      bar = [int (double d, char c)]
            {
                 return 1;
            };
      return 1;
}
```

The translation proceeds as follows:

```
class context0 implements int_function_double_array_char_
{
    double [] x ;
    char  y ;
    int_function_double_array_char_  bar ;

    public context0(double[] x, char y,
                     int_function_double_char_ bar)
    {
        this.x=x ;
        this.y=y ;
        this.bar=bar ;
    }

    public int apply(double d, char c)
    {
        return function_foo.lambda0(x, y, bar, d, c);
    }
} // end class context0

class function_foo implements int_function_double_array_char_
{
    public static int foo(double[] x, char y)
    {
        int_function_double_char_  bar = null;
        bar = new context0(x, y , bar) ;
        return 1 ;
    }

    public int apply(double[] x , char y)
    {
        return foo(x,y);
    }
    static public int lambda0(double[] x, char y,
                              int_function_double_char_  bar,
                              double d, char c)
    {
        return 1 ;
    }
}   //  end class function_foo
```

And for translation in the case of a non-static member function consider,

```
class A
{
    public void baz(String[] args)
    {
          [int (double[] , char)] bar = null;
          bar = [int (double[] x, char y)]
                {
                    return 1;
                };
    }
}
```

which is translated into:

```
class A
{
    public void baz(String[] args)
    {
        int_function_double_array_char_ bar = null;
        bar = new context1(this, args, bar ) ;
    }

    class context1 implements int_function_double_array_char_
    {

        A receiver;
        String [] args ;
        int_function_double_array_char_  bar ;

        public context1 (A receiver, String[] args,
                         int_function_double_array_char_  bar)
        {
            this.receiver=receiver;
            this.args=args ;
            this.bar=bar ;
        }

        public int apply(double[] x, char y)
        {
            return receiver.lambda1( args, bar, x, y );
        }
    } // end class context1

    public int lambda1(String[] args, int_function_double_array_char_  bar,
                       double[] x , char y)
    {
        return 1 ;
    }
} // end class A
```

3.3 Operator Overloading

All operators are assigned names in Java/MP. Operator overloading in Java/MP is done by declaring a function with the corresponding name in the global scope. This technique is used in Leda, as well as other languages such as Python. There are a couple of restrictions. The assignment operator `=`, the arrow operator `<-`, and the conditional assignment operator `?:` cannot be overloaded. Also, the default meanings assigned by Java for operators cannot be changed. So arithmetic operators cannot be overloaded on `int`s, the complement operator `!` cannot be overloaded on `booleans` etc. The current prototype implementation does not allow overloading of operators on any primitive types. This is an implementation imposed restriction and not a language defined restriction. Refer to Appendix A. for names assigned to operators. Following code shows the binary operator plus (+) overloaded on `Integer` arguments.

```
Integer plus (Integer x, Integer y)
{
    return new Integer(x.intValue() + y.intValue());
}
```

This is translated into Java as follows:

```
class function_plus implements Integer_function_Integer_Integer_
{
    public static Integer plus(Integer x, Integer y)
    {
        return new Integer(x.intValue() + y.intValue()) ;
    }

    public Integer apply(Integer x, Integer y)
    {
        return plus(x,y);
    }
}
```

This translation technique is no different than the one used for global functions. A statement like,

```
System.out.println(i+j); // i and j are Integer types.
```

is simply translated into the following if operator + has been overloaded for `Integer` arguments:

```
System.out.println( function_plus.plus(i,j) );
```

3.4 Function Overloading

Function overloading is standard in Java. Java/MP allows global functions also to be overloaded. One point to keep in mind is that the compiler cannot differentiate between a two implementations of a function that solely differ in the parameter passing mechanism. That is, we cannot overload function `foo` as follows:

```
int foo(int i, int j)
{
   //…..whatever
}
```

```
        int foo(int @ i, int j)
        {
           // … whatever
        }
```

The reason for this restriction is due to the ambiguity that arises when a method call such the following is made:

```
        foo(3,4);
```

It is not clear which one of the two methods should be invoked.

3.5 Scheme for generating interface names

As noted previously we generate one interface for every function type. The scheme used by the prototype implementation is rather simple but generates a unique string for each function type. Basically the interface name consists of the return type followed by "_function_" followed by parameter types suffixed with "_".
So for the function type

```
              int wow(String s, double d)
```

we have the string *`int_function_String_double_`*. When arrays are found, each dimension is simply replace by "array_". So for

```
        int wow(String[] s, int j[][])
```

we get *`int_function_String_array_int_array_array_`*. When parameters are passed by name, instead of only suffixing the type name with an underscore, we prefix the type name with "_Thunk_" and also suffix it with an underscore. So for:

```
        int wow(int i, boolean @ b)
```

we get the string int_function_int__Thunk_boolean_.
Similarly the scheme just extends to parameters or return values of function type. For:

```
        [int(boolean)] wow( [int(boolean)] func )
```

we obtain *`int_function_boolean__function_int_function_boolean_`*.

3.6 Pass by Name

The purpose of introducing pass by name in Java/MP is twofold. First, evaluation of pass by name arguments is delayed until the point an expression is used rather than the point at which the function call is made. Second, allow changes made by the callee to a by name parameter to reflect back in the caller. When an argument is to be passed by name, we simply wrap the argument in a class (better known as a *thunk*) and use an instance of this thunk as the actual argument.

The Java/MP standard library provides a class `__Thunk` that specifically serves this purpose. It is defined as follows:

```
abstract class __Thunk   {
   abstract public Object get();
   public void set(Object rhs) {};
};
```

When passing an expression by name, a class deriving from `__Thunk` is created. This class will implement the abstract `get` method so that the expression can be (re)evaluated at any point necessary. Since it is the responsibility of this class to implement the `get` method, class `__Thunk` declares it as `abstract`. The `set` method on the other hand is implemented only if the argument being passed is a variable and not a general expression. Thus the `set` method is not declared to be `abstract`. In addition to implementing the `get` and `set` methods, the `__Thunk` derived class will also hold any context required by the expression for which it was created.

Consider the function `printInt` that accepts one parameter by name.

```
static void printInt( int @ i )
{
    System.out.println(i);
}
```

This is the translated into:

```
static void printInt( __Thunk i )
{
    System.out.println( ( (Integer)(i.get()) ).intValue() ) ;
}
```

First we notice that the parameter type has been changed to `__Thunk` and the `get` method is invoked when the value of `i` is required. Second, there is an automatic conversion between `int` and `Integer` when invoking `get`. This is because the get method is declared to return an `Object` and primitive types like `int`, `float` etc. are not `Objects`. The thunk created for the any argument passed to `printInt` will need to create an `Integer` from the `int` value obtained by evaluating the expression that it holds. In the case of object types obviously no such conversions are needed. Consider the a function `foo` which invokes `printInt`:

```
static void foo()
{
   int i = 5;
   int j = 4;
   printInt(i+10);
}
```

Since the expression `i+10` needs to be converted into a thunk, the following class is created in the global scope:

```
class Expression24Context extends __Thunk
{
    int  i;
    int  j;
    Expression24Context( int  i , int  j )
    {
        this.i=i;
        this.j=j;
    }
    public Object get()
    {
```

```
                return new Integer ( i + 10 );
            }
            /* no set function needed */
        }
```

Notice the context variables being saved in `Expression24Context`. Both `i` and `j` are stored since they are the only variables that could possibly be needed by the expression being passed by name. An optimization step could actually notice the fact that `j` is not really needed and hence keep it out of `Expression24Context`.

The call to `printInt` is then simply translated into:

```
        printInt( new Expression24Context(i, j) ) ;
```

3.7 Context Reuse

When a variable is passed by name we create thunk for it as described above. If variable is assigned a new value inside the callee, this change has to reflect in the caller. So, once a thunk has been created for a variable, we need to make sure that the every occurrence of that variable henceforth should be substituted with a call to the `get` or `set` method of the thunk's instance. So consider the following code:

```
        addOne(int @ i)
        {
           i=i+1;
        }

        foo()
        {
           int i=4;
           addOne(i);
           System.out.println(i); // should print 5 !
        }
```

Since change in the value of `i` has to reflect beyond the call to `addOne()`, the argument passed to the print statement should in fact be replaced with a call to the `get` method of the thunk created for `i`.

```
        foo()
        {
           int i=4;
           Expression45Context ectx45 = new Expression45Context(i);
           addOne(ectx45);
           System.out.println((Integer)ectx45.get());
        }
```

3.8 Logic programming

Logic programming is implemented using a combination of all the features mentioned above, a new data type called **relation** and the new **arrow** operator. A rather simplified way of understanding a relation is to think of it as an extension of boolean. It really represents a function that returns a boolean. In addition, it also takes another relation as argument. Since we would like to pass relations as arguments and return them from other functions, the internal representation implements this function as part of a class. The type relation is defined in the standard library as follows:

```
        interface relation
        {
```

```
        public boolean apply(relation continuation);
    }
```

The library includes an important and handy but simple relation called `trueRelation` as follows:

```
    class trueRelation implements relation
    {
       public boolean apply(relation continuation)
       {
             return true;
       }
    }
```

As the name suggest this relation always returns true and pays no attention to its argument. Throwing away the argument enables `trueRelation` to be effectively used as a device to halt the recursion introduced in the definition of `relation`. `trueRelation` also proves to be very useful in converting relations into booleans. Similar to the `trueRelation`, the library also includes a `falseRelation` that always returns `false`.

3.8.1 Arrow Operator

Java/MP introduces the new **arrow operator `<-`**, which is used as a reversible assignment operator. That is, it can undo an assignment if some future relation does not hold true. The Java implementation of the arrow operator in the library is shown in Appendix B.6. The Java/MP version presented below is shorter and simpler to understand. The following code is conceptually equivalent to the Java version.

```
    relation arrow(Object @ left, final Object right)
    {
        return  [boolean (relation continuation)]
                {
                    Object save = left;
                    left = right;
                    if(continuation)
                          return true;
                    left = save;
                    return false;
                };
    }
```

As we can see in the Java/MP version, the arrow operator takes two `Objects` as parameters and returns a `relation` (a function). The first parameter is taken by name in order to allow any changes to reflect back in the calling routine. The function returned by the arrow operator will save the value of `left`, assign `right` to `left` and then check if `continuation` holds true. If so, the new value of `left` is allowed to be permanent and `true` is returned, else the initial value of `left` is restored and `false` is returned to indicate a failure of the assignment.

3.8.2 Operators && and ||

In addition to the `<-` operator, Java/MP relies heavily on the default implementations for the `&&` and `||` operators to provide support for logic style programming. The Java implementations for the operators `&&` and `||` are shown in Appendix B. sections B.7 and B.8 respectively. Once again for simplicity reasons, let us consider their Java/MP equivalent implementations.

```
    relation and (relation left, relation @ right)
    {
```

```
   return [boolean (relation continuation)]
          {
              return left(  [boolean(relation useless)]
                            {
                                return right(continuation);
                            }
                        );
          };
}

relation or (relation left, relation @ right)
{
    return  [boolean (relation continuation)]
            {
                if(left(continuation))
                    return true;
                return right(continuation);
            };
}
```

Both the binary `&&` and `||` operators are overloaded to accept two `relation` arguments and return a `relation`. The second parameter in both cases is passed by name to allow short circuit evaluation similar to C/C++. So the `&&` operator can disregard its second argument in case the first argument evaluates to `false`.

Implementation for the `&&` operator is slightly more complex than `||`. It involves two lambda functions. One that is returned and another that is passed as the continuation argument to `left`. The function that is passed to `left` basically invokes `right`. Note that when any relation is invoked by passing it another relation, the former gets effectively converted into a boolean. This is because a relation is really a function that returns a boolean. Alternatively, if we pass `right` directly to `left`, `right` gets evaluated prior to `left` and that is not what we want. The above technique lets `left` decide whether or not `right` should be invoked. This technique, though a little tricky, is cleverly devised to fit smoothly into the flow of logic programming. Refer to [Budd, 1995] for a more detailed explanation of the workings.

The `||` operator's implementation is rather straightforward. It simply returns a function (a relation) that will execute `right` only if the `left` returns `false`. The variable `continuation` is passed as argument to both left and `right`. It is important to note that this incredibly simple implementation for the `||` operator in conjunction with the `<-` operator provides the backtracking mechanism needed for logic programming. Briefly, the `||` operator tries to pursue one of two alternatives. If the `left` alternative fails it tries `right`. The function returned by operators `<-`, `&&`, the `eq` function or another relation making use of any of the former will typically form the `left` and `right` arguments. A backtracking mechanism in general, needs to perform two important tasks. The first is to try another alternative (if any) when the first fails. The second is to undo changes (made to variables while pursuing the first alternative) before moving on to the second alternative. The first task is performed by the `||` operator and second by the `<-` operator.

3.8.3 Relational *if* and *while*

A `while` loop or an `if` statement is given special treatment when the test expression is of type `relation`. First consider the following while statement:

```
while(relExpr)
{
   System.out.println(…)
}
```

First, the body of the while loop is given the same treatment as a lambda function. Hence a function called `lambda34` (suffixed with an arbitrary number) is created, within the enclosing class, containing the same body as the while loop. All the context information needed is passed as parameters. A context class `context34` is created and instantiated to hold the context variables just before the while statement. This is responsible for invoking the `lambda34` and passing parameters to it.

Second, `relExpr` is converted into a boolean by invoking its `apply` method and passing a relation to it as argument. This relation argument is specially constructed to first cause the instance of `context34` to invoke the `lambda34` and then return `false`. `false` is returned in order to fool the backtracking mechanism into believing that `relExpr` has failed (returned false) when `relExpr` actually succeeded, and thus resume its search for other alternatives. The reason for this behavior is due to the fact that `relExpr` will invoke its continuation every time it succeeds, Consequently `lambda34` will be executed and `false` returned. When `relExpr` truly returns `false`, the execution exits the `while` loop since `relExpr` will not invoke its continuation.

The translation proceeds as follows:

```
// lambda for the loop body
public void lambda34(type1 local1, type2 local2)
{
   System.out.println(…) ;
}

// context to invoke the loop body
class context34 implements void_function_void
{
  // all locals to be held as context
  type1 local1;
  type2 local2;

  public context34 (type1 local1, type2 local1)
  {
     this.local1=local1 ;
     this.local2=local2 ;
  }

  // invoke lambda34 and pass context info
  public void apply()
  {
     lambda34(local1, local2);
  }
}

// the while loop
final context34 _ctx34= new context34(local1, local2);
relExr.apply( new relation()
              {
                public boolean apply(relation continuation)
                   {
                      _ctx34.apply();
                      return false;
                   }
              }
             );
```

The translation process for the relational `if` statement is a little simpler since we do not need to fool the backtracking mechanism to process all alternatives. The only difference is in the in second step, the argument passed as continuation while converting `relExpr` into a boolean is simply an instance of trueRelation. Thus for

```
if(relExpr)
{
   System.out.println(…)
}
```

The first step of the translation stays identical to the while loop's translation. But the second step results in the following.

```
if( relExpr.apply(new trueRelation()) == true )
    _ctx34.apply();
```

Chapter 4. Java/MP Example Programs and Techniques

4.1 Iteration

Many useful styles of iteration are described in chapter 5 of [Budd, 1995]. This section discuses two additional iteration techniques. The first proves useful in enabling logical style iteration over existing object-oriented containers. The second one is an attempt to improve upon the traditional object-oriented iteration to fit somewhat better in the multiparadigm framework.

4.1.1 Logic Style Iteration for traditional data structures

Container type data structures in logic and functional languages often have recursive definitions. In Java/MP we could define a list of integers recursively as follows:

```
List
{
    Integer element;
    List next;
}
```

Such recursive data structures are very useful for functional and logic style programs. Logic style iteration on such containers can be implemented as follows.

```
List
{
    relation items(Integer @ value)
    {
      return eq(value,element)
                  || (next!=null) && next.items(value);
    }
}
```

Iterating through the list can then simply done as follows,

```
List myList;
//…add  Integers to list
Integer x=null;
while( myList.items(x) )
{
   System.out.println(x);
}
```

In the object-oriented world, we often encounter data being held in arrays or some other non-recursive structure. Consider the simple case of a string. Strings are almost always implemented as an array of characters. In Java/MP we could define it as:

```
Class mpString
{
   Character[] string;

   public mpString(String value)
   {
      // initialize string with values from value
```

```
        }
        public mpString(char[] value)
        {
           // initialize string with values from value
        }
     }
```

It would be nice to derive `mpString` from the Java `String` class, but unfortunately `String` is declared as `final` which means that it cannot be subclassed. Since the logical iteration technique described above relied on the recursive definition of `List`, we are unable to reuse the same for implementing the `items` method in `mpString`. The items method for `mpString` can be implemented as follows:[9]

```
     public relation items(final Character @ c)
     {
        return new relation()
            {
               int i=0;  // index
               public boolean apply(final relation continuation)
               {
                 if( i >= string.length )
                      return  (c!=null)
                                 && continuation.apply(new trueRelation());

                 return eq(c,string[i++])
                             && continuation.apply(new trueRelation())
                        || apply(continuation) ;
               }
            };
     }
```

The `items` method returns a handcrafted relation object containing a member variable `i`. This variable is used as an index to the current element. Thus every invocation to items will return a relation that contains its own index. The `apply` method unifies `c` with the currently indexed character, increments the index and executes `continuation`. The backtracking comes into play when the unification fails causing the recursive call to apply. Since the index has been incremented already during unification, each successive call to `apply` causes unification with the next element in sequence.

This technique is useful to allow logical iteration over container classes such as the Java `Vector` or some other custom container. Simply create a sub-class that implements the `items` method using the above technique. For arrays, a simple integer index suffices; others may find iterators or more complex variations useful. The Java translation (manually optimized for readability) is presented in Appendix C.

4.1.2 Iterator Objects Variation:

Often functions can be written very succinctly if they can iterate over the elements of a recursively defined container.

[9] A problem with the translation process does not allow this code for the items method to be generated correctly. The call to apply in the return statement will incorrectly invoke of the apply method of the context created for the second argument to the || operator leading to endless recursion. A work around would be to create a nested class deriving from relation, and qualify the apply method name with the class name when invoking. The Java code in the appendix is edited manually to resolve the correct apply function.

```
// test if all elements are less than 5 in the list
boolean allLessThanFive_a(List numbers)
{
  if(numbers==null)
      return true;
  return numbers.element.intValue()<5 && allLessThanFive(numbers.next)
}
```

Languages such as C++ and Smalltalk use iterator objects pervasively for such tasks. Iterators based code would be similar to:

```
boolean allLessThanFive_b( List.iterator numbers )
{
   if(numbers==null)
       return true;
   if(numbers.value().intValue()<5)
   {
      numbers.next();
      return allLessThanFive(numbers);
   }
   return false;
}
```

The iterator for class `List` could be implemented as an inner class of List.

```
public class iterator
{
   int index=0;
   public Integer value()
   {
     return elements[index]; // elements[] defined in outer class List
   }
   public void next()
   {
     ++index;
   }
   public void prev()
   {
     --index;
   }
}
```

Such an iterator can be a problem when used in conjunction with the backtracking. The problem being that once the iterator is moved the backtracking mechanism cannot easily restore the iterator to an earlier position.

An improved iterator can be implemented as follows:

```
public class iterator
{
     final int index;
     public iteratator(int index)
     {
       return this.index=index;
     }
     public Integer value()
     {
       return elements[index]; // elements[] defined in outer class
     }
```

```
            public iterator next()
            {
             ( i+1 < elements.length ) ? new iterator(i+1) : null;
            }
            public void prev()
            {
             return new iterator(index-1);
            }
       }
```

Each call to `next()` will instantiate a new iterator pointing the next element in sequence. This allows it to work better with backtracking. Since there is no state change involved within the iterator, there is no need to restore an iterator to a previous value. Another important property is that an iterator pointing to the last element in the sequence will return null when `next()` is invoked. This iterator also allows the simpler `allLessThanFive_a` style code to be used within the iterators based `allLessThanFive_b` function. The examples in following sections demonstrate the usefulness of this iterator when used in conjunction with backtracking.

4.2 Non Deterministic Finite Automata (NFA)

Finite state machines accept string input and reports success or error if the string matches the pattern defined by some regular expression. Consider building the NFA for the following regular expression.

`((ab)*ba)|b`

The graphical representation for the corresponding NFA is shown in Figure 1.

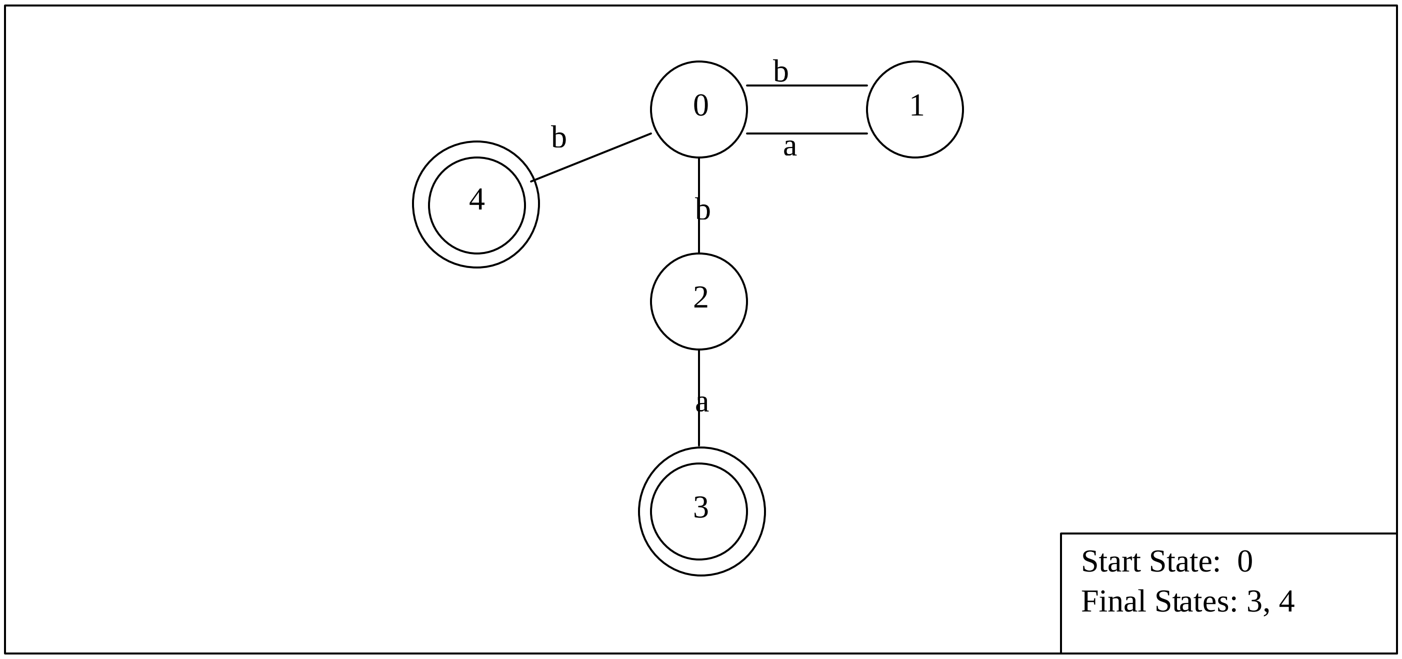

Figure 1: NFA for ((ab)*ba) | b

A finite state automaton has one start state, one or more final states, and transitions between states. The transition from state 0 to state 1 on input character b in the above NFA can be represented by the following relational expression:

```
        eq( state1, zero ) && eq( input, A ) && eq(state2, one)
```

where `zero`, `one` represent integers 0 and 1 respectively and `A` represents the character ‘a’. All transitions in the NFA can be abstracted by a user defined relation called `transition` using disjunctions (“or” expressions) as follows:

```
        Class NFA
        {
           static relation transition(Integer state1,
                                       Character input, Integer @ state2)
            {
                Character A = new Character('a'), B = new Character('b');
                Integer zero = new Integer(0), one = new Integer(1),
                        two  = new Integer(2), three = new Integer(3),
                        four = new Integer(4);

                return
                    eq(state1, zero) && eq(input, A) && eq(state2, one)
                ||  eq(state1, zero) && eq(input, B) && eq(state2, two)
                ||  eq(state1, zero) && eq(input, B) && eq(state2, four)
                ||  eq(state1, one)  && eq(input, B) && eq(state2, zero)
                ||  eq(state1, two)  && eq(input, A) && eq(state2, three) ;
            }
        }
```

Given a state `state1` and a character `input`, relation `transition` will produce a value for `state2` such that there exists a transition form state1 to state2. Java/MP backtracking proves useful again here since there can be more than one possible state2. If there exist no transition starting from state1 on the given input, state2 will remain null.

Since NFA’s can have multiple final states we implement a boolean member function `final` in class `NFA` to test for final states.

```
        static boolean relation final(Integer State)
        {
                if (State.intValue()==3 || State.intValue()==4)
                   return true;
                return false;
        }
```

The relation `run` will simulate the NFA over any input string by iterating over the string, one character at a time and checking if a transition exists from the current state on that character.

```
        static relation run( Integer startState, mpString.iterator input )
        {
          if( final(startState) && input==null )
               return true;    // Input accepted
          Integer nextState = null;
          return  (input != null)
                  && transition( startState, input.value(), nextState )
                  && run(nextState, input.next()) ;
        }
```

When input string is exhausted and a final state has been reached, the `run` method will return `true` indicating success. This verificatio n is done by the first `if` statement. The return statement ensures that the input string has not run out and simulates the NFA over the entire input by recursively applying `transition`. The iterator technique described in section 4.1.2 is used here. Each recursive call to `run` receives a new iterator pointing to the next character. The run method, with the help of backtracking, attempts to pursue all possible paths until the input is fully consumed and final state is reached. The non-determinism in the NFA may cause it to pursue an incorrect path, in which case backtracking will back up on the consumed input and from the current state to try alternative paths. Since there was no state change within the iterator, rolling back the recursive call to `run` is sufficient to back it up.

To see how this works, let us consider the a sample input string “abb”.

```
mpString str = new mpString(“abb”);
if( run(new Integer(0), str.getIterator()) )  // 0 = start state
    System.out.println("Accepted");
else
    System.out.println("Rejected");
```

Variable `str` is initialized with the string to be tested and we invoke `run` with the start state 0 and an iterator to the first character in the string (obtained by invoking `getIterator()`). Given this iterator, the `run` method is able to iterate over the string one character at a time. The `run` method then invoke the `transition` method to check if a transition from state 0 to any state (`nextState`) exists for the first character ‘a’. If so, the iterator to the next character is obtained and `run` calls itself with `nextState` as the start state. This repeats the process over each of the remaining characters. It is interesting too see how backtracking occurs here. When the first two characters have been consumed and the run method is considering the third character ‘b’, it takes one of the two possible values for ‘nextState’ namely 2 and 4. It pursues 2 first and then realizes on the next recursive call to `run`, that the input has run out and final state has not been reached. This causes the run method to return to a previous invocation of itself where the iterator pointed to the last character b. The backtracking mechanism executes the `transition` method to be again in order to find an alternate transition from state 0 to another state on character b. This time 4 is chosen as the next state and a another recursive call to `run` proceeds. But since a final state has now been reached and the input has been consumed completely, the run method completely returns from all recursive calls and returns true to the if statement. Thus `System.out.println("Accepted")` is executed indicating success.

4.3 Telephone Numbers to Words

The next example enumerates all the words that can be formed from a telephone number. We first need a data structure that maps each digit to one or more characters. Since indexing into this data structure will always be done using an integer, a vector of strings will suffice. We can define the mapping as follows.

```
final mpVector map = new mpVector();
map.addElement(new mpString(" "));      // 0
map.addElement(new mpString(" "));      // 1
map.addElement(new mpString("ABC"));    // 2
map.addElement(new mpString("DEF"));    // 3
map.addElement(new mpString("GHI"));    // 4
map.addElement(new mpString("JKL"));    // 5
map.addElement(new mpString("MNO"));    // 6
map.addElement(new mpString("PRS"));    // 7
map.addElement(new mpString("TUV"));    // 8
map.addElement(new mpString("WXY"));    // 9
```

The class mpVector used above inherits form the Java Vector and provides the iteration techniques described in sections 4.1.1 and 4.1.2 mpStrings are stored in the vector, and thus we can easily iterate over the different characters for each digit.

The method gen shown below, generates one word for a given phone number. When used as argument to the relational while loop it generates all words. Class mpString is used to represent a phone number and mpString.iterator is used to iterate over it. The word generated is the regular Java String.

```
static relation gen(mpString.iterator digits, String @ word)
{
  if(digits == null)
       return true;
  mpString s=(mpString)map.elementAt(characterToInt(digits.value()));
  Character c = null;
  return
      s.items(c)
      && (word <- charCat(word,c))
      && gen(digits.next(), word );
}
```

Since the String class does not provide any direct function to append Characters, we use the charCat method defined in Appendix D. to append a Character. The method characterToInt also defined in Appendix D. , is used to convert a character representing a number into an int. The return statement is a conjunction of three expressions. The first picks a character c for the first digit in the phone number, the second appends c to word and the third expression performs the previous two steps on the remaining digits.

To generate all words for a phone number, we simple invoke gen as follows:

```
mpString phNumber = new mpString(“5332”);
String word = null;
while( gen( str.getIterator(), word ) )
{
    System.out.print( word );
    System.out.print(", ");
}
```

Chapter 5. Implementation Notes

A prototype implementation for the Java/MP compiler was developed as part of this project to test the language and verify the translation techniques described in [Budd, 2000]. This chapter gives a brief overview of the implementation and also the lists the known limitations.

5.1 Implementation Details

The compiler is implemented as a command line executable that processes a single Java/MP source file and produces Java source code as output. Input and output files are specified as command line parameters. The Java source produced as output is fed to a Java compiler to produce byte codes. The compiler is written using flex version 2.5.4 and GNU bison 1.2.8. The source consists of about 8600 lines of code and approximately six months were spent in developing the compiler. An existing Java 1.1 grammar [Dmitri, 1998] for flex and bison was used as a starting point. Changes and additions to the grammar were made to suit the code generation better and accommodate the following features

1. Global functions and operator overloading
2. Defining functions as expressions,
3. Declaring variables of functions type
4. `instanceof` operator for unboxing.
5. Arrow operator
6. Specifying parameters to be passed by name

When executed, the compiler first parses the standard library followed by the input source file specified on the command line. When processing the standard library, code generation is turned off and only symbol tables are updated. The standard library actually has two implementations. Once in the *std.jmp* file and again in a file called *std.java.* The first provides the implementation in Java/MP and the second file in pure Java. The former is simply used to update symbol tables with information regarding the standard library while the later is included into the output Java file. This dual implementation is done for two reasons. First, the Java implementation is edited to remove any redundancies and improve readability and efficiency. Second, since *std.jmp* is only used to build symbol tables, it serves as a place to add dummy class definitions for classes implemented in the Java standard packages. This provides a convenient work around for the limitation described in section 5.3.1.

5.2 Testing

The compiler implementation has not undergone a rigorous and formalized testing process. The primary goal in developing the compiler was to demonstrate that our ideas for Java/MP were in fact realistic rather than implement an industrial strength compiler. However the implementation has been put through some amount of testing as noted below to make sure its quality is acceptable for our research purposes.

The symbol table was first tested independent of the parser by using scripts that added entries into the symbol table, performed lookups and printed all symbol table content. The aim was to verify if the symbol table behaved as expected and to determine any memory leaks. Borland Code Guard was used to monitor runtime memory usage for detecting memory leaks. All the editing and debugging was done in Borland C++ Builder. After integrating the symbol table into the parser, testing was done using sample Java/MP programs to ensure that modifications to the symbol table did not introduce new problems. The system was again subjected to memory leak detection. The lexer and parser are known to have memory leaks that are difficult to fix due to the inherent nature of flex and bison[10]. The code generation was also tested using sample Java/MP programs. Testing the code generation was done in two ways. First, simple programs were developed to check if specific language features were handled correctly. Second, a few small but more realistic programs were written to detect problems when combining language features. However, the compiler has not been exhaustively tested using test suites that are typically used by commercial compiler vendors nor have any formal testing tools been used expect for detecting memory leaks.

5.3 Limitations

This section describes some known limitations in the prototype Java/MP compiler implementation.

5.3.1 No Class Loader

Java classes are distributed as binary packages. Imported packages need to be loaded and parsed in order to fill symbol tables with information necessary to parse the program. The prototype implementation does not implement a class loader. Dummy class definitions can be provided in the *std.jmp* file as described in section 5.1 for this purpose.

5.3.2 No forward references

Forward references to classes or functions are not allowed. Thus all functions and classes must be defined before they are used. Consequently circular referencing of the following nature is not allowed, and there is no simple work around.

```
void foo( )
{
    bar();
}
void bar()
{
   foo();
}
```

5.3.3 Declaring arrays

Java allows declaring the dimensions for an array both after the type name and variable name. For example:

```
int [] x;
int [] x,y;    // x = [] and y = []
int y[];
int[] a, b[];  // a = [] and b = [][]
```

The implementation us unable to handle the third and fourth forms of array declaration well in certain cases during code generations. No error is reported. It is preferable to use the first and second forms.

5.3.4 Initializers

In Java, any variable can be initialized at the point of declaration. For example;

```
int i = 20;
```

[10] Memory is often explicitly allocated for tokens that are found by the lexer. Intuitively, this memory should be freed each time the next token is to be found. However, it is dangerous to free the memory used by the current token since the lexer often tends to look one token beyond the beyond the current. It is difficult to determine at which point a lookahead will be done by the lexer so that the memory allocated for the current token can be deallocated.

```
int[] y = {1,2,3,4} , z = {5,6,7};
String[] s = {new String("hello"), new String("world")};
```

The implementation allows such declarations but it does not type check such the initializations. Hence if the initialization expression needs processing during code generation errors may be introduced in the output code. Such initializations typically work fine since they are copied verbatim into the output. But sometimes, similar initializations in Java/MP involving function variables or any other that requires special handling may causes bad code to be generated. So it is recommend to use the following kind of code:

```
[int(char)] func = null;
func = [int(char c )] { return 4; };
```

instead of:

```
[int(char)] func = [int(char c )] { return 4; };
```

Also always try to initialize local variables with `null` when declaring them. This reason being that contexts to hold values of all locals could get created at any point. These contexts cannot be initialized correctly unless the local variables have already been initialized. Consequently the Java compiler may report an error in the generated output.

5.3.5 Assignment Operators and Overloading

Assignment operators like +=, `*`= etc. cannot be overloaded on function types.

```
// operator \=
int divideAssign([int()] foo, [int()] bar) // not allowed
{
 //..
}
```

The compiler does not notice or this report error.

5.3.6 Chaining Assignments

Avoid chaining assignments as follows if any of the variables involved in the chain is of by name type.

```
x = y = 23; // problem if y is a byName parameter
```

This is because, both the `get` and `set` methods for the thunk created for y need to be called here at the same point. The `set` method has to be called to assign `23` to `y` and then the `get` method to assign the value of `y` to `x`.

5.3.7 Private Public and Protected members

The compiler treats all members as public.

5.3.8 Inheritance and Interfaces

The algorithm used to resolve a variable or function name does a search to find the declaration in the following order.

1. Current scope
2. Base class scope
3. Outer scope

Base interfaces are not looked up.

5.3.9 Resolving functions only by name

Java/MP allows named functions to assigned to variables using the name of the function.

```
[int(double)] funcVar = null;
funcVar = functionName;
```

The compiler uses the very first function it finds with that name. This can be a problem if the function is overloaded.

5.3.10 Bitwise complement operator ~

The return type of the bit-wise complement operator is assigned a default typ e of `int`.

5.3.11 Conditional operator ?:

The return type of the conditional operator is same as the expression returned when the condition is true.

5.3.12 Arithmetic Expression type

The type for a binary arithmetic expression is determined from the first argument unless the operator is overloaded on for the types involved.

```
3.4 * 5     // type is float
5 * 3.5     // type is int
```

5.3.13 Type conversions and promotions

The Java language specification [Java, 2000] devotes an entire chapter documenting all automatic type conve rsions and promotions. The prototype performs only the following promotions and conversions:

`boolean` is converted into a `relation` or vice versa whenever one is used in place of the other.
An instance of child class can be used where a base class instance is required.
`null` can be use where any object type is required.

All other conversions and promotions need explicit use of casting.

5.3.14 Operator Overloading not allowed on primitive types

In Java/MP, operators can be given new meanings by overloading only if Java does not already define the meaning of an operator on those types. Effectively system defined meanings cannot be changed. The current implementation does allow overloading operators on primitive types. Assignment operator, arrow operator and the cond itional assignment operator are not overloadable.

Chapter 6. Future Work and Conclusion

6.1 Future Work

The current prototype compiler implementation was built as a tool for evaluating and verifying many of the ideas related to implementation we had in mind for Java/MP. Thus many simplifying assumptions have been made as noted in section 5.3. But availability of a solid compiler is essential for every programming language. As Nicklaus Wirth notes,

"In practice, a programming language is as good as its compiler(s)"

The current prototype Java/MP implementation could use several major improvements.

1) The first and most important feature would perhaps be the inclusion of Java class file parser. This would allow packages to be imported into Java/MP programs without jumping through the hoops described in section 5.1.
2) Since the current implementation produces Java source code as output, every Java/MP program undergoes compilation twice. First by the Java/MP compiler then by a Java compiler. By producing Java bytecode directly instead of source code, we can obviate the need for the second compilation.
3) A debugger that can handle relations, by name parameters and higher order functions would prove to be an indispensable tool.
4) A more optimized code generation strategy can reduce the run time memory requirements significantly. For example, when creating a context for a lambda function, it would be preferable to include in the context, only those variables that are actually used by the lambda. This strategy would also be useful when creating thunks for by name parameters. When only a variable (and not an expression) is passed by name, only that one variable can be stored within the thunk.

6.2 Conclusion

Adding multiparadigm features to Java is certainly a feasible idea. Java/MP combines the logic, object oriented and functional paradigms in a manner that allows the programmers to mix and match the paradigms seamlessly. Java/MP adds very few extensions and maintains much of the feel of Java. There is no doubt about the usefulness of a multiparadigm language as a tool for problem solving and is demonstrated by the Leda book [Budd, 1995]. Java/MP brings these benefits to a much wider audience by using Java as its foundation.

It is evident from the translation techniques that Java/MP programs will typically have greater memory requirements than pure Java solutions for the same problem. It is difficult to estimate how execution efficiency will be affected without conducting formal tests. No testing or benchmarks have been conducted to accurately asses the runtime efficiency and memory overhead of Java/MP programs.

Appendix A.   Java/MP Operators

All operators are assigned names in Java/MP. The following is a list of all the operators

Binary operators	
>	greater
<	less
<=	lessEqual
>=	greaterEqual
==	equals
!=	notEqual
&&	and
||	or
&	bitAnd
|	bitOr
^	bitExOr
+	plus
-	minus
*	multiply
/	divide
%	remainder
<<	leftShift
>>	rightShift
>>>	unsignedRightShift
+=	plusAssign
-=	minusAssign
*=	mutiplyAssign
/=	divideAssign
&=	andAssign
|=	orAssign
^=	exOrAssign
%=	divideAssign
<<=	leftShiftAssign
>>=	rightShiftAssign
>>>=	unsignedRightShiftAssign

Unary operators	
!	not
~	bitComplement
++	prefixIncrement
++	postfixIncrement
--	prefixDecrement
--	postfixDecrement
+	unaryPlus
-	unaryMinus

Non-overloadable operators	
<-	arrow
?:	conditionalAssign
=	assign

Appendix B.   Java/MP Standard Library

This section provides the implementations of all the important functions and classes provided by the Java/MP standard library

B.1 __instofHelper class

```
class __instofHelper
 {

     public static boolean returnTrue(Object val)
     {
        return true;
     }
     public static boolean returnTrue(boolean val)
     {
        return true;
     }
     public static boolean returnTrue(char val)
     {
        return true;
     }
     public static boolean returnTrue(byte val)
     {
        return true;
     }
     public static boolean returnTrue(short val)
     {
        return true;
     }
     public static boolean returnTrue(int val)
     {
        return true;
     }
     public static boolean returnTrue(long val)
     {
        return true;
     }
     public static boolean returnTrue(float val)
     {
        return true;
     }
     public static boolean returnTrue(double val)
     {
        return true;
     }
 }
```

B.2 Relation

```
interface relation
{
   public boolean apply(relation continuation);
}
```

B.3 trueRelation and falseRelation

```
class trueRelation implements relation
{
   public boolean apply(relation continuation)
      {
         return true;
      }
}

class falseRelation implements relation
{
     public boolean apply(relation continuation)
     {
       return false;
     }
}
```

B.4 booleanAsRelation function

```
class function_booleanAsRelation
                    implements relation_function_Thunk_boolean_
{
    public static relation booleanAsRelation( final __Thunk value )
    {
        return new relation ( )
        {
            public boolean apply( relation continuation )
            {
                return ( ((Boolean)(value.get())).booleanValue() )  &&
                    continuation.apply( new trueRelation() ) ;
            }
        } ;
    }

    public relation apply(final __Thunk value)
    {
        return booleanAsRelation(value);
    }
}
```

B.5 __Thunk class

```
abstract class __Thunk
{
    abstract public Object get();
    public void set(Object rhs) { };
};
```

B.6 <- Operator

```
class function_arrow implements relation_function___Thunk_Object_
{
    public static relation arrow(final __Thunk left, final Object right)
    {
        return  new relation()
            {
                public boolean apply (relation continuation)
                {
                    Object save = left.get();
                    left.set(right);
                    if(continuation.apply(new trueRelation()))
                        return true;
                    left.set(save);
                    return false;
                }
            };
    }
    public relation apply(__Thunk left, Object right)
    {
        return arrow(left,right);
    }
}
```

B.7 && Operator

```
class function_and implements relation_function_relation___Thunk_relation_
{
   public static relation and( final relation left, final __Thunk right)
   {
       return new relation()
             {
                public boolean apply(final relation continuation)
                {
                    return left.apply(
                           new relation()
                              {
                                public boolean apply(relation f)
                                {
                                   relation r = (relation) right.get();
                                   return r.apply(continuation);
                                }
                              }
                    );  // end left.apply(..)
                }
       };  // end return new relation()
   }
   public relation apply( relation left, __Thunk right)
   {
        return and(left,right);
   }
}
```

B.8 || Operator

```
class function_or implements relation_function_relation___Thunk_relation_
{
    public static relation or( final relation left, final __Thunk right )
    {
        return  new relation()
                {
                    public boolean apply(relation continuation)
                    {
                        if(left.apply(continuation))
                            return true;
                        relation r = (relation) right.get();
                        return r.apply(continuation);
                    }
                };
    }
    public relation apply( relation left, __Thunk right )
    {
        return or(left,right);
    }
}
```

B.9 Miscellaneous interfaces

```
interface relation_function_Thunk_Object_Object_
{
    relation apply( __Thunk p1, Object p2 );
}

interface relation_function_relation___Thunk_relation_
{
    relation apply( relation p1, __Thunk p2);
}
```

Appendix C.   Java code for Items Method

```
class mpString
{
   public relation items( final __Thunk c )
   {
      return new relation ( )
           {
              int i = 0 ;
              public boolean apply( final relation continuation )
              {
                final relation This=this; //used to resolve this apply method
                if (i >= string.length )
                     return c.get()!=null
                            && continuation.apply(new trueRelation());

                ExprCtx _ectx_ = new ExprCtx ( continuation );
                return function_or.or(
                      function_and.and(function_eq.eq(c,string[i++]),_ectx_),
                         new __Thunk()
                             {
                                public Object get()
                                {
                                   return new relation()
                                         {
                                             public boolean apply(relation f)
                                             {
                                               return This.apply(continuation);
                                             }
                                         };
                                 }
                              }
                        ).apply(new trueRelation());
              }
           };
   }

   private  class ExprCtx extends __Thunk
   {
        relation  continuation;
        ExprCtx( relation  continuation)
        {
            this.continuation=continuation;
        }
        public Object get()
        {
            return continuation;
        }
        public void set( Object rhs)
        {
            continuation = (relation)rhs;
        }
   }
}
```

Appendix D. Utility Functions

```
/* convert c to int */
int characterToInt(Character c)
{
    return Character.digit(c.charValue(),10);
}

/* catenate c to s */
String charCat(String s ,Character c)
{
    if(c!=null )
    {
        if(s==null)
            return c.toString();
        return s.concat(c.toString());
    }
    return null;
}
```

Appendix E. Java/MP grammar

E.1 Lexical Analyzer Specification as flex Input

```
%e 1600
%n 800
%p 5000

Separator        [\(\)\{\}\[\]\;\,\.]
Delimiter1       [\=\>\<\!\~\?\:\+\-\*\/\&\|\^\%]
HexDigit         [0-9a-fA-F]
Digit            [0-9]
OctalDigit       [0-7]
TetraDigit       [0-3]
NonZeroDigit     [1-9]
Letter           [a-zA-Z_]
AnyButSlash      [^\/]
AnyButAstr       [^\*]
BLANK            [ ]
BLK              [\b]
TAB              [\t]
FF               [\f]
ESCCHR           [\\]
CR               [\r]
LF               [\n]
UniEsc           [\1b]

OctEscape1       [\\]{OctalDigit}
OctEscape2       [\\]{OctalDigit}{OctalDigit}
OctEscape3       [\\]{TetraDigit}{OctalDigit}{OctalDigit}
OctEscape        ({OctEscape1}|{OctEscape2}|{OctEscape3})

Escape           [\\]([r]|[n]|[b]|[f]|[t]|[\\]|[\']|[\"])
ULetter          ({Letter}|{UniEsc})
Identifier       {ULetter}({ULetter}|{Digit})*

Comment1         [\/][\*]({AnyButAstr}|[\*]{AnyButSlash})*[\*][\/]
Comment2         [\/][\/].*
Comment          ({Comment1}|{Comment2})

Dimension        [\[]({CR}|{LF}|{FF}|{TAB}|{BLK}|{BLANK}|{Comment})*[\]]

IntSuffix  ([l]|[L])
DecimalNum       {NonZeroDigit}{Digit}*{IntSuffix}?
OctalNum         [0]{OctalDigit}*{IntSuffix}?
HexNum           [0]([x]|[X]){HexDigit}{HexDigit}*{IntSuffix}?
IntegerLiteral   ({DecimalNum}|{OctalNum}|{HexNum})

Sign             ([\+]|[\-])
FlSuffix         ([f]|[F]|[d]|[D])
SignedInt        {Sign}?{Digit}+
Expo             ([e]|[E])
ExponentPart     {Expo}{SignedInt}?
Float1           {Digit}+[\.]{Digit}+?{ExponentPart}?{FlSuffix}?
Float2           [\.]{Digit}+{ExponentPart}?{FlSuffix}?
Float3           {Digit}+{ExponentPart}{FlSuffix}?
Float4           {Digit}+{FlSuffix}
FloatingPoint    ({Float1}|{Float2}|{Float3}|{Float4})

AnyChrChr        [^\\']
AnyStrChr        [^\\"]
Character        [\']({Escape}|{OctEscape}|{AnyChrChr})[\']
String           [\"]({Escape}|{OctEscape}|{AnyStrChr})*[\"]
```

```
Numeric         ({IntegerLiteral}|{FloatingPoint})
Literal         ({Numeric}|{Character}|{String})

"true"          { return BOOLLIT;}
"false"         { return BOOLLIT;}

{Separator}     { return yytext[0];}
{Delimiter1}    { return yytext[0];}
{Dimension}     { return OP_DIM;}

"=="            { return OP_EQ;}
"<="            { return OP_LE;}
">="            { return OP_GE;}
"!="            { return OP_NE;}
"||"            { return OP_LOR;}
"&&"            { return OP_LAND;}
"++"            { return OP_INC;}
"--"            { return OP_DEC;}
">>"            { return OP_SHR;}
"<<"            { return OP_SHL;}
">>>"           { return OP_SHRR;}
"+="            { return ASS_ADD;}
"-="            { return ASS_SUB;}
"*="            { return ASS_MUL;}
"/="            { return ASS_DIV;}
"&="            { return ASS_AND;}
"|="            { return ASS_OR;}
"^="            { return ASS_XOR;}
"%="            { return ASS_MOD;}
"<<="           { return ASS_SHL;}
">>="           { return ASS_SHR;}
">>>="          { return ASS_SHRR;}
"<-"            { return ASS_ARROW;}

"abstract"      { return ABSTRACT;}
"do"            { return DO;}
"implements"    { return IMPLEMENTS;}
"package"       { return PACKAGE;}
"throw"         { return THROW;}
"boolean"       { return BOOLEAN;}
"double"        { return DOUBLE;}
"import"        { return IMPORT;}
"private"       { return PRIVATE;}
"throws"        { return THROWS;}
"break"         { return BREAK;}
"else"          { return ELSE;}
"inner"         { return INNER;}
"protected"     { return PROTECTED;}
"transient"     { return TRANSIENT;}
"byte"          { return BYTE;}
"extends"       { return EXTENDS;}
"instanceof"    { return INSTANCEOF;}
"public"        { return PUBLIC;}
"try"           { return TRY;}
"case"          { return CASE;}
"final"         { return FINAL;}
"int"           { return INT;}
"finally"       { return FINALLY;}
"interface"     { return INTERFACE;}
"return"        { return RETURN;}
"void"          { return VOID;}
"catch"         { return CATCH;}
"float"         { return FLOAT;}
"long"          { return LONG;}
"short"         { return SHORT;}
"volatile"      { return VOLATILE;}
```

```
"char"            { return CHAR;}
"for"             { return FOR;}
"native"          { return NATIVE;}
"static"          { return STATIC;}
"while"           { return WHILE;}
"class"           { return CLASS;}
"new"             { return NEW;}
"super"           { return SUPER;}
"const"           { return CONST;}
"generic"         { return GENERIC;}
"null"            { return JNULL;}
"switch"          { return SWITCH;}
"continue"        { return CONTINUE;}
"goto"            { return GOTO;}
"operator"        { return OPERATOR;}
"synchronized"    { return SYNCHRONIZED;}
"default"         { return DEFAULT;}
"if"              { return IF;}
"outer"           { return OUTER;}
"this"            { return THIS;}
"@"               { return BYNAME;}
{Identifier}      { return IDENTIFIER;}

{DecimalNum}      { return LITERAL;}
{OctalNum}        { return LITERAL;}
{HexNum}          { return LITERAL;}

{Float1}          { return LITERAL;}
{Float2}          { return LITERAL;}
{Float3}          { return LITERAL;}
{Float4}          { return LITERAL;}

{Character}       { return LITERAL;}
{String}          { return LITERAL;}

{CR}              {}
{LF}              {}
{FF}              {}
{TAB}             {}
{BLK}             {}
{BLANK}           {}

{Comment}         {}
```

E.2 Parser Specification as Bison input

```
TypeSpecifier     : SimpleTypeSpecifier
                  | FunctionTypeSpecifier
;
FunctionTypeSpecifier           : '[' TypeSpecifier '(' TypeSpecifierList ')' ']'
                                | '[' TypeSpecifier '(' TypeSpecifierList ')' ']' Dims
                                | '[' TypeSpecifier '(' ')' ']'
                                | '[' TypeSpecifier '(' ')' ']' Dims
;
SimpleTypeSpecifier   : TypeName
                      | TypeName Dims
;
TypeSpecifierList  : TypeSpecifierList ',' TypeSpecifier
                   | TypeSpecifier
;
TypeName    : PrimitiveType
            | QualifiedName
;
```

```
ClassNameList    : QualifiedName
                 | ClassNameList ',' QualifiedName
;
PrimitiveType    : BOOLEAN
                 | CHAR
                 | BYTE
                 | SHORT
                 | INT
                 | LONG
                 | FLOAT
                 | DOUBLE
                 | VOID
;
SemiColons       : ';'
                 | SemiColons ';'
;
CompilationUnit   : ProgramFile
;
ProgramFile      : PackageStatement ImportStatements TypeAndFunctionDeclarations
                 | PackageStatement ImportStatements
                 | PackageStatement TypeAndFunctionDeclarations
                 | ImportStatements TypeAndFunctionDeclarations
                 | PackageStatement
                 | ImportStatements
                 | TypeAndFunctionDeclarations
;
PackageStatement : PACKAGE QualifiedName SemiColons
;
TypeAndFunctionDeclarations          : GlobalFunctionDeclaration
                                     | TypeDeclarationOptSemi
                                     | TypeAndFunctionDeclarations TypeDeclarationOptSemi
                                     | TypeAndFunctionDeclarations GlobalFunctionDeclaration
;
GlobalFunctionDeclaration    : MethodDeclaration
;
TypeDeclarations             : TypeDeclarationOptSemi
                             | TypeDeclarations TypeDeclarationOptSemi
;

TypeDeclarationOptSemi    : TypeDeclaration
                          | TypeDeclaration SemiColons
;
ImportStatements             : ImportStatement
                             | ImportStatements ImportStatement
;
ImportStatement  : IMPORT QualifiedName SemiColons
                 | IMPORT QualifiedName '.' '*' SemiColons
;
QualifiedName   : IDENTIFIER
                  | QualifiedName '.' IDENTIFIER
;
TypeDeclaration   : ClassHeader '{' FieldDeclarations '}'
                  | ClassHeader '{' '}'
;
ClassHeader      : Modifiers ClassWord IDENTIFIER Extends Interfaces
                 | Modifiers ClassWord IDENTIFIER Extends
                 | Modifiers ClassWord IDENTIFIER              Interfaces
                 |           ClassWord IDENTIFIER Extends Interfaces
                 | Modifiers ClassWord IDENTIFIER
                 |           ClassWord IDENTIFIER Extends
                 |           ClassWord IDENTIFIER              Interfaces
                 |           ClassWord IDENTIFIER
;
Modifiers        : Modifier
                 | Modifiers Modifier
;
```

```
Modifier              : ABSTRACT
                      | FINAL
                      | PUBLIC
                      | PROTECTED
                      | PRIVATE
                      | STATIC
                      | TRANSIENT
                      | VOLATILE
                      | NATIVE
                      | SYNCHRONIZED
;
ClassWord             : CLASS
                      | INTERFACE
;
Interfaces            : IMPLEMENTS ClassNameList
;
FieldDeclarations              : FieldDeclarationOptSemi
                               | FieldDeclarations FieldDeclarationOptSemi
;
FieldDeclarationOptSemi         : FieldDeclaration
                                | FieldDeclaration SemiColons
;
FieldDeclaration               : FieldVariableDeclaration ';'
                               | MethodDeclaration
                               | ConstructorDeclaration
                               | StaticInitializer
                               | NonStaticInitializer
                               | TypeDeclaration
;
FieldVariableDeclaration       : Modifiers TypeSpecifier VariableDeclarators
                               |           TypeSpecifier VariableDeclarators
;
VariableDeclarators            : VariableDeclarator
                               | VariableDeclarators ',' VariableDeclarator
;
VariableDeclarator             : DeclaratorName
                               | DeclaratorName '=' VariableInitializer
;
VariableInitializer            : Expression
                               | '{' '}'
                               | '{' ArrayInitializers '}'
;
ArrayInitializers     : VariableInitializer
                      | ArrayInitializers ',' VariableInitializer
                      | ArrayInitializers ','
;
MethodDeclarationHeader                : Modifiers TypeSpecifier  MethodDeclarator Throws
                                       | Modifiers TypeSpecifier  MethodDeclarator
                                       |           TypeSpecifier  MethodDeclarator Throws
                                       |           TypeSpecifier  MethodDeclarator
;
MethodDeclaration              : MethodDeclarationHeader MethodBody
;
MethodDeclarator               : DeclaratorName '(' ParameterList ')'
                               | DeclaratorName '(' ')'
                               | MethodDeclarator OP_DIM
;
ParameterList                  : Parameter
                               | ParameterList ',' Parameter
;
Parameter             : TypeSpecifier DeclaratorName
                      | SimpleTypeSpecifier BYNAME DeclaratorName
                      | FINAL TypeSpecifier DeclaratorName
                      | FINAL SimpleTypeSpecifier BYNAME DeclaratorName
;
DeclaratorName         : IDENTIFIER
```

```
                                | DeclaratorName OP_DIM
;
Throws        : THROWS ClassNameList
;
MethodBody          : Block
                    | ';'
;
ConstructorDeclaration      : Modifiers ConstructorDeclarator Throws Block
                            | Modifiers ConstructorDeclarator        Block
                            |           ConstructorDeclarator Throws Block
                            |           ConstructorDeclarator        Block
;
ConstructorDeclarator       : IDENTIFIER '(' ParameterList ')'
                            | IDENTIFIER '(' ')'
;
StaticInitializer   : STATIC Block
;
NonStaticInitializer   : Block
;
Extends             : EXTENDS TypeNameList
;
TypeNameList        : TypeName
                    | TypeNameList ',' TypeName
;
Block               : '{' LocalVariableDeclarationsAndStatements '}'
                    | '{' '}'
;
LocalVariableDeclarationsAndStatements    : LocalVariableDeclarationOrStatement
            | LocalVariableDeclarationsAndStatements LocalVariableDeclarationOrStatement
;
LocalVariableDeclarationOrStatement         : LocalVariableDeclarationStatement
                                            | Statement
;
LocalVariableDeclarationStatement           : TypeSpecifier VariableDeclarators ';'
                                            | FINAL TypeSpecifier VariableDeclarators ';'
;
Statement           : EmptyStatement
                    | LabelStatement
                    | ExpressionStatement ';'
                    | SelectionStatement
                    | IterationStatement
                    | JumpStatement
                    | GuardingStatement
                    | Block
;
EmptyStatement : ';'
;
LabelStatement   : IDENTIFIER ':'
                    | CASE ConstantExpression ':'
                    | DEFAULT ':'
;
ExpressionStatement         : Expression
;
SelectionStatement          : IF '(' SelectionOrLoopExpression ')' Statement
                            | IF '(' SelectionOrLoopExpression ')' Statement ELSE Statement
                            | SWITCH '(' SelectionOrLoopExpression ')' Block
;
SelectionOrLoopExpression           : Expression
;
IterationStatement          : WHILE '(' SelectionOrLoopExpression ')' Statement
                            | DO Statement WHILE '(' SelectionOrLoopExpression ')' ';'
                            | FOR '(' ForInit ForExpr ForIncr ')' Statement
                            | FOR '(' ForInit ForExpr        ')' Statement
;
ForInit             : ExpressionStatements ';'
                    | LocalVariableDeclarationStatement
```

```
                          | ';'
;
ForExpr                   : Expression ';'
                          | ';'
;
ForIncr   : ExpressionStatements
;
ExpressionStatements          : ExpressionStatement
                              | ExpressionStatements ',' ExpressionStatement
;
JumpStatement                 : BREAK IDENTIFIER ';'
                              | BREAK             ';'
                              | CONTINUE IDENTIFIER ';'
                              | CONTINUE           ';'
                              | RETURN Expression ';'
                              | RETURN           ';'
                              | THROW Expression ';'
;
GuardingStatement             : SYNCHRONIZED '(' GuardingExpression ')' Statement
                              | TRY Block Finally
                              | TRY Block Catches
                              | TRY Block Catches Finally
;
GuardingExpression            :  Expression
;
Catches                   : Catch
                          | Catches Catch
;
Catch     : CatchHeader Block
;
CatchHeader               : CATCH '(' TypeSpecifier IDENTIFIER ')'
                          | CATCH '(' TypeSpecifier ')'
;
Finally   : FINALLY Block
;
PrimaryExpression             : QualifiedName
                              | NotJustName
;
NotJustName               : SpecialName
                          | NewAllocationExpression
                          | ComplexPrimary
;
ComplexPrimary            : '(' Expression ')'
                          | ComplexPrimaryNoParenthesis
;
ComplexPrimaryNoParenthesis       : LITERAL
                                  | BOOLLIT
                                  | ArrayAccess
                                  | FieldAccess
                                  | MethodCall
;
ArrayAccess               : QualifiedName '[' Expression ']'
                          | ComplexPrimary '[' Expression ']'
;
FieldAccess               : NotJustName '.' IDENTIFIER
                          | RealPostfixExpression '.' IDENTIFIER
;
MethodCall                : MethodAccess '(' ArgumentList ')'
                          | MethodAccess '(' ')'
;
MethodAccess              : ComplexPrimaryNoParenthesis
                          | SpecialName
                          | QualifiedName
;
SpecialName               : THIS
                          | SUPER
                          | JNULL
;
ArgumentList        : Expression
                          | ArgumentList ',' Expression
;
IdentifierList         :  QualifiedName
                          |  IdentifierList ',' QualifiedName
;
NewAllocationExpression   : PlainNewAllocationExpression
                                  | QualifiedName '.' PlainNewAllocationExpression
;
PlainNewAllocationExpression              : ArrayAllocationExpression
                                                      | ClassAllocationExpression
                                                      | ArrayAllocationExpression '{' '}'
                                                      | ArrayAllocationExpression '{' ArrayInitializers '}'
                                                      | AnonymousClass
;
AnonymousClass    : AnonymousClassHeader '{' FieldDeclarations '}'
                            | AnonymousClassHeader '{' '}'
;
AnonymousClassHeader                : NEW TypeName '(' ArgumentList ')'
                                                | NEW TypeName '(' ')'
;
ClassAllocationExpression             : NEW TypeName '(' ArgumentList ')'
                                                | NEW TypeName '('              ')'
;
ArrayAllocationExpression             : NEW TypeName DimExprs Dims
                                                | NEW TypeName DimExprs
                                                | NEW TypeName Dims
;
DimExprs             : DimExpr
                          | DimExprs DimExpr
;
DimExpr :             '[' Expression ']'
;
Dims       : OP_DIM
              | Dims OP_DIM
;
PostfixExpression              : PrimaryExpression
                                      | RealPostfixExpression
;
RealPostfixExpression        : PostfixExpression OP_INC
                                      | PostfixExpression OP_DEC
;
UnaryExpression               : OP_INC UnaryExpression
                                      | OP_DEC UnaryExpression
                                      | ArithmeticUnaryOperator CastExpression
                                      | LogicalUnaryExpression
;
LogicalUnaryExpression     : PostfixExpression
                                      | LogicalUnaryOperator UnaryExpression
;
LogicalUnaryOperator        : '~'
                                      | '!'
;
ArithmeticUnaryOperator   : '+'
                                      | '-'
;
CastExpression                : UnaryExpression
                                      | '(' PrimitiveTypeExpression ')' CastExpression
                                      | '(' ClassTypeExpression ')' CastExpression
                                      | '(' Expression ')' LogicalUnaryExpression
;
PrimitiveTypeExpression   : PrimitiveType
                                      | PrimitiveType Dims
;
```

```
ClassTypeExpression       : QualifiedName Dims
;
MultiplicativeExpression  : CastExpression
                          | MultiplicativeExpression '*' CastExpression
                          | MultiplicativeExpression '/' CastExpression
                          | MultiplicativeExpression '%' CastExpression
;
AdditiveExpression        : MultiplicativeExpression
                          | AdditiveExpression '+' MultiplicativeExpression
                          | AdditiveExpression '-' MultiplicativeExpression
;
ShiftExpression           : AdditiveExpression
                          | ShiftExpression OP_SHL AdditiveExpression
                          | ShiftExpression OP_SHR AdditiveExpression
                          | ShiftExpression OP_SHRR AdditiveExpression
;
RelationalExpression      : ShiftExpression
                          | RelationalExpression '<' ShiftExpression
                          | RelationalExpression '>' ShiftExpression
                          | RelationalExpression OP_LE ShiftExpression
                          | RelationalExpression OP_GE ShiftExpression
                          | RelationalExpression INSTANCEOF SimpleTypeSpecifier
                          | RelationalExpression INSTANCEOF SimpleTypeSpecifier '(' IdentifierList ')'
;
EqualityExpression        : RelationalExpression
                          | EqualityExpression OP_EQ RelationalExpression
                          | EqualityExpression OP_NE RelationalExpression
;
AndExpression             : EqualityExpression
                          | AndExpression '&' EqualityExpression
;
ExclusiveOrExpression     : AndExpression
                          | ExclusiveOrExpression '^' AndExpression
;
InclusiveOrExpression     : ExclusiveOrExpression
                          | InclusiveOrExpression '|' ExclusiveOrExpression
;
ConditionalAndExpression : InclusiveOrExpression
                          | ConditionalAndExpression OP_LAND InclusiveOrExpression
;
ConditionalOrExpression   : ConditionalAndExpression
                          | ConditionalOrExpression OP_LOR ConditionalAndExpression
;
ConditionalExpression     : ConditionalOrExpression
                          | ConditionalOrExpression '?' Expression ':' ConditionalExpression
;
AssignmentExpression      : ConditionalExpression
                          | UnaryExpression AssignmentOperator AssignmentExpression
                          | FunctionExpression
;
FunctionExpression        : FunctionExpressionHeader MethodBody
;
FunctionExpressionHeader      : '[' TypeSpecifier '(' ParameterList ')' ']'
                              | '[' TypeSpecifier '(' ')' ']'
;
AssignmentOperator        : '='
                          | ASS_ARROW
                          | ASS_MUL
                          | ASS_DIV
                          | ASS_MOD
                          | ASS_ADD
                          | ASS_SUB
                          | ASS_SHL
                          | ASS_SHR
                          | ASS_SHRR
                          | ASS_AND
```

```
                                   | ASS_XOR
                                   | ASS_OR
;
Expression    : AssignmentExpression
;
ConstantExpression          : ConditionalExpression
;
```

Bibliography

[Budd, 1995]	Timothy Budd (Oregon State University). Multiparadigm Programming in Leda, Addison Wesley. 1995.
[Budd, 2000]	Timothy Budd (Oregon State University), The design of Java/MP, Oregon State University. March 30, 2001
[Diomidis, 1994]	Diomidis Spinellis, Sophia Drossopoulou, and Susan Eisenbach. An object model for multiparadigm programming. In Dennis Kafura, Greg Lavender, and Siva Challa, editors, *OOPSLA '94 Workshop on Multi-Language Object Models*, Oregon, USA, October 1994.
[Dmitri, 1998]	Dmitri Bronnikov. A free yacc-able Java grammar. August 1998. http://home.inreach.com/bronikov/grammars/java.html
[Eiffel, 1992]	Bertrand Meyer. Eiffel The Language, Prentice Hall 1992.
[Gödel, 1994]	Patricia Hill, John Lloyd. The Gödel programming Language, MIT press, 1994
[Haskell, 1999]	Simon Thompson. The Craft of Functional programming, Addison Wesley,1999.
[Java, 2000]	James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java Language Specification, Second Edition. Addison-Wesley
[Odersky, 1997]	Martin Odersky (University of Karlsruhe) and Philip Wadler (University of Glasgow). Pizza into Java: Translating Theory into Practice, Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997.
[Ruby,2000]	David Thomas, Andrew Hunt, Dave Thomas. Programming Ruby, A Pragmatic Programmers Guide, Addison Wesley, December 2000.