2m11.2934.8

Université de Montréal

A simplified Java: J

Par

Xinjian Xue

Département d'informatique et de recherche opérationnelle

Faculté des Arts et des Sciences

Mémoire présenté à la Faculté des études supérieures

En vue de l'obtention du grade de maîtrise

En informatique

Octobre, 2001

© Xinjian Xue, 2001

QA
76
U54
2002
V.005

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé:

A simplified Java: J

présenté par

Xinjian Xue

a été évalué par un jury composé des personnes suivantes:

Jian-Yun Nie

président-rapporteur

Jean Vaucher

directeur de recherche

Pierre Poulin

membre du jury

Mémoire accepté le: 7 décembre 2001

Résumé

Dans ce mémoire, nous considérons la conception et la mise en place d'une meilleure forme de Java qui pourrait être employée pour enseigner et programmer des algorithmes sans devoir recourir au pseudo-code. Nous appelons ce langage J. Il modifie la forme externe de Java, mais maintient sa sémantique interne: elle utilise le même code d'octet et la même machine virtuelle.

J retire quelques irritants de Java: il automatise en partie la conversion de type pour réduire au minimum l'utilisation explicite de « cast »; il utilise également une déclaration simplifiée pour la méthode *main*. J utilise la notation mathématique ou algorithmique standard autant que possible. Par exemple, J utilise ": = " pour l'affectation et " = " pour l'égalité. Dans J, les opérateurs classiques de comparaison (>, > =, =, etc.) s'appliquent aux objets de n'importe quelle classe qui met en application l'interface *Comparable*. De même, la notation d'indexation (c.-à -d. a[i]) peut être employée pour accéder à des éléments de chaînes de caractères et de vecteurs au lieu d'avoir à utiliser une méthode comme *a.elementAt(i)*. J utilise également la syntaxe algorithmique pour les énoncés *if* et *while*. Comme le Pascal, J essaye d'imposer une certaine mesure de structure aux déclarations de classe. Pour séparer clairement des attributs d'*instance* des attributs (statiques) de *classe* et pour éviter la répétition, J emploie le délimiteur *static* comme séparateur après quoi toutes les déclarations (*main* y compris) sont considérées comme statique.

Le changement le plus complexe de J est le relâchement de l'algorithme de sélection de méthodes *surchargées*. J emploie une notion de distance entre les types primitifs pour choisir la version la plus appropriée d'une méthode. L'étude détaillée de ce dispositif a cependant soulevé quelques problèmes fondamentaux.

Le compilateur pour J est mis en application à l'aide de plusieurs outils existants: JavaCC, le compilateur de compilateur de Sun; BCEL, une bibliothèque de classes pour manipuler le code d'octet; et Espresso, un compilateur complet pour Java 1.0. Nous présentons brièvement ces outils et donnons des exemples démontrant comment les utiliser. Notre modification principale au compilateur comporte l'ajout d'une passe particulière à J: un visiteur pour transformer tous les noeuds spécifiques à J dans l'arbre syntaxique en noeuds de Java standard. Nous ajoutons également un formatteur au compilateur, qui peut sortir l'équivalent en Java d'un programme de J. Ceci illustre le travail du compilateur, aide dans la mise au point et fournit une base pour la comparaison de J et de Java.

Pour terminer, nous faisons quelques comparaisons pour tester si nous avons réalisé nos buts de clarté et de brièveté. Les essais objectifs comparant la longueur des programmes équivalents prouvent que les programmes de J sont toujours plus courts que la version de Java: avec des gains de 8.8% du nombre de jetons et de 7.8% du nombre de caractères en moyenne.

Mots clés : construction compilateur, compilateur-compilateur, indexation, surcharge, cast, Espresso, JavaCC.

Abstract

In this thesis, we consider the design and implementation of a better form of Java that could be used to teach and program algorithms without having to resort to pseudo-code. We call this language J. It modifies the external form of Java but retains its internal semantics: it uses the same byte-code and the same virtual machine.

J removes some Java irritants: it automates some type casting to minimize the explicit use of cast expressions; it also uses a simplified main method declaration. J uses standard mathematical or algorithmic notation as much as possible. For example, J uses ":=" for assignment and "=" for value. In J, the classic comparison operators (>, >=, =, etc.) apply to objects of any class which implements the *Comparable* interface. Similarly, indexing notation (i.e., a[i]) can be used to access elements of both Strings and Vectors instead of having to use methods like a.*elementAt(I).* J also uses algorithmic syntax for *if* and *while* statements. Like Pascal, J tries to impose some measure of structure on class declarations: to clearly separate *instance* attributes from *class* (static) attributes and to avoid repetition, J uses static as a delimiter after which all declarations (including main) are considered static.

The most complex change that J introduces is the relaxing of the method selection algorithm of Java. It uses the notion of distance between primitive types to allow the closest version of a method to be selected. Detailed study of this feature brought out some fundamental problems.

The compiler for J is implemented using several existing tools: JavaCC, Sun's compiler-compiler; BCEL, a library of classes to manipulate byte code; and Espresso, a complete compiler for Java 1.0. We introduce briefly these and give examples demonstrating how to use them. Our major modification involves the addition of a J-specific compiler phase in the form of a visitor to transform all the J-specific syntax features on the syntax tree into standard Java. We also add a pretty printer to the compiler, which can output the Java equivalent of a J program. This illustrates the work of the compiler, helps in debugging and provides a basis for the comparison of J and Java.

Finally, we do some comparisons to test whether we achieved our goals of clarity and brevity. Objective tests comparing the length of equivalent programs show that J programs are always shorter than the Java version: with gains of 8.8% fewer tokens and of 7.8% fewer characters on average.

Key words: compiler construction, compiler-compiler, indexing, overload, type casting, Espresso, JavaCC.

Table of contents

Acknowledgements

I would like to express my sincere thanks to my supervisor, Professor Jean Vaucher without whose help this thesis would not have been possible. Acknowledgement of his guidance on my study and his constant encouragement that helped me to overcome my difficulties can not be adequately recognized in a few words.

I would also want to take this opportunity to thank all the professors who have taught me courses and the supporting staff in the department. They have made my study at DIRO a pleasant and productive experience.

Finally, thanks go to my wife and my parents for their constant support that accompanied me to go through the good time and the tough time, without which I could not have reached this far.

Chapter 1

Introduction

Java is the newest development in computer languages. It incorporates many fine features but this does not mean that it is perfect. In this thesis, we try to show how Java could be improved while retaining its good qualities.

From its first public release in 1994, Java has rapidly become a very popular programming language. Java is associated with the World Wide Web -- and the Web's global scale -- and it has applications near and far, from smart cards to the 2001 Mars Lander. There are hundreds of books on Java, specialist magazines, and web sites. Java is now taught in hundreds of universities. Java is clearly a mainstream phenomenon.

Java is an obvious example of a successful blend of new and old features. With its object orientation, parallelism, and extensive Internet library, Java has the right blend of features for modern application developers to enhance their productivity. Java improves the security of programs with garbage collection, the elimination of explicit pointers, and more stringent type checking and is suitable for security sensitive Internet applications. Geared to the exploding growth of the internet, its widespread diffusion was helped by the fact that it came out just at the right time to be incorporated into WWW browsers and distributed free all over the planet.

Java is not totally revolutionary; it is based on C and has kept many features of that language. Far from being a problem, this allows C programmers to switch easily to Java. Thus both the old and new features contribute equally to the success of Java.

Though Java has achieved great success, both commercial and academic, it means neither that it is perfect as a computer language, nor that it could not be improved. Java has several problems.

Java support of formatted I/O is worse than that of its predecessors (C and Pascal). Simple tasks such as reading an integer or printing a number with a specified number of decimals require several lines of code and use of auxiliary classes. The Java syntax based on C is more hermetic than that of Algol based languages. This problem is compounded by the introduction of additional keywords, which reflect the new features of the language as well as the objective to be totally "object oriented". As a result, Java programs tend to be overly wordy and algorithm descriptions in Java are much less readable than those in other languages like Pascal or in pseudo-code. This stands in contrast to previous developments where new languages allowed clearer expression.

The lack of clarity of Java is especially a problem when it comes to the teaching of both programming and algorithms. However, there are so many useful modern features in the language: objects, graphics, parallelism, distributed applications, security, and portability, that Java is the language of choice in many universities. There are two solutions to bridge the gap: either design a complete new language, or extend Java in some way. Obviously, the latter is easier and it benefits from any progress in Java.

We choose the second solution, that is, to extend Java by modifying its external form and retaining the Java internals: JVM and Java byte code. We design a language for this purpose and call it J.

The compiler for J is based on an open source research project named Espresso [Espresso98]. Espresso is a full Java compiler for Java 1.0.2 written in Java at Boston University. Its main purpose is to serve as a workbench for students interested in applying their theoretical knowledge of programming languages and compiler technologies and using modern compiler construction tools like JavaCC to construct a compiler for a modern language.

J improves on Java in the following ways. It uses Pascal-like syntax to make the program more readable and structured; it introduces different operators for the primitive value equality test and the object identity test to clearly distinguish those two tests; it allows use of comparison operators on objects; it extends the use of indexing operator [] on classes String and Vector to simplify program syntax; and it generalizes Java's method selection algorithm. Many of these changes are relatively simple to implement but the more important work goes into automating some type conversions to reduce the need for the explicit type cast operation.

The rest of the thesis is organized as follows: chapter 2 gives a brief overview over Java's evolution, its new language concepts and its advantages and disadvantages; chapter 3 is the design of the language J; chapter 4 is a survey of some compiler construction tools and the project "Espresso"; chapter 5 explains how the J compiler is implemented; chapter 6 tells us if J achieves our design goal by running some test programs and doing some comparisons with plain Java; and chapter 7 concludes the whole thesis.

Chapter 2

Survey of the Java Language

Java is an object oriented programming language developed in the early 1990's by James Gosling at Sun Microsystems. Originally, it was designed for embedded consumer electronic application and the language was called Oak. In 1994, Guy Steele, and Bill Joy joined James Gosling; they re-targeted the language to the Internet and renamed it Java. As the premier Internet language with an interpreter in every web browser, Java gained rapid acceptance. As of November 1997, when Java was 900 days old, there were 900,000 active programmers [Javax]. By comparison, it took C++ ten years to achieve the same acceptance level.

In this chapter, we will first have a look at Java's evolution, and then examine more closely its design and its language concepts, commenting on its advantages and disadvantages.

2.1 Evolution of Java versions

Since its birth, from Java 1.0 to the latest version Java 1.3, Java has always been improving and evolving. It can help us better understand the Java language itself if we consider the various stages of their evolutions.

- **Java 1.0**

Java 1.0 was the first official release, however it contained some critical security related bugs that were fixed in versions 1.0.1 and 1.0.2. Generally, when Java 1.0 is referred to, it means Java 1.0.2, the first stable release.

Java 1.0.2 had 8 packages and 211 classes or interfaces [Java10]. It laid down the main structure of the language, the only important language component missing was the inner class. This thesis is based on Java 1.0.2, for which a full compiler was available.

- **Java 1.1**

Java 1.1 had 22 packages and 477 classes and interfaces [Java11]. It introduced many new features listed in table 2.1, in addition to many refinements in areas like AWT, networking, I/O, etc.

Table 2.1 New features introduced in Java 1.1

Inner Class	Internationalization	Security package
JAR File Format	Remote Method Invocation	Object Serialization
Reflection	Java Database Connectivity	Java Native Interface

With the inner class, mainly used to create adapter classes, Java 1.1 completes the Java syntax design. The internationalization feature allows the development of locale sensitive applications and the use of Unicode characters. It makes Java suitable for language sensitive applications and helped make Java popular in different countries all over the world no matter what language is spoken there. The Java Security package is designed to allow fine-grained control over individual operation for various classes of applications. Remote Method Invocation (RMI) enables the programmer to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines in a simple way. Reflection enables Java code to discover information about the fields, methods, and constructors of loaded classes, and to operate on objects whose class is unknown at compile time. Native Interface allows Java programs to call native methods and thus interface with and reuse legacy systems.

- **Java 1.2**

Java 1.2 had 59 packages and 1524 classes and interfaces [Java12]. Java 1.2 adds the following new features: Swing GUI toolkit and Collections framework. Java Swing GUI toolkit extends the original Abstract Window Toolkit (AWT) by adding a comprehensive set of graphical user interface class libraries. Java Swing GUI Components are written in Java, without window-system-specific code. This facilitates a customizable look and feel without relying on the native windowing system, and simplifies the deployment of applications. It is a major improvement over AWT. Collections framework is a unified architecture for representing and manipulating data structure. Collections (sometimes called containers) are simply objects that group multiple elements into a single unit and are used to store, retrieve, and manipulate data.

- **Java 1.3**

The latest Java version is Java 1.3 and it has 76 packages and 1840 classes and interfaces [Java13]. As usual, in addition to introducing many refinements, Java 1.3 added libraries oriented to distributed applications: Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI) over IIOP, CORBA ORB, Java IDL compiler idlj, etc.

2.2 Concepts of Java

Java builds upon tried and true programming language concepts. First it is object-oriented and embodies features from Smalltalk, Simula, C++, and Modula. For syntax, Java uses that of C or C++, which is familiar to a large number of users; this has proved helpful for Java's immediate and widespread acceptance.

Java, leaving C++'s legacy aside, borrows more from Pascal than C with regards to type safety. Java does not allow assignment between incompatible types. Sometimes this can seem like an unnecessarily severe restriction, but the payback is worthwhile: by preventing defects at initial coding, it saves lots of the time spent later on debugging.

Furthermore, a virtual machine verifier checks at loading time that an object is well formed and contains no security violation.

Learning from the problems in C++, Java uses lazy binding for methods. This means that, unlike in C++, code for method calls is not hard-wired into the compiled program. Thus it is not necessary to recompile all code that uses a library when the library changes. This and the removal of the need for header files as in C and C++ means that programming environments for Java can dramatically speed up the "round-trip engineering cycle": from debug to edit to compile and back to debug.

2.2.1 Portability and Java Virtual Machine (JVM)

Java programs can run on any platform that has a Java Virtual Machine implemented. Portability has traditionally meant writing source code that conforms to industry standards and then adapts this code to the variety of platforms that support the programming language. Each port is a significant new testing effort because of the differences between platforms. The goal for Java is to write and compile Java programs once and run the resulting "machine" code on every platform without change.

The Java Virtual Machine, or JVM, is an abstract computer that runs compiled Java programs. The JVM is "virtual" because it is generally implemented in software on top of a hardware platform and operating system. All Java programs are compiled for the JVM. Therefore, the JVM must be implemented on a particular platform before compiled Java programs will run on that platform. However, it is easier to write an interpreter (VM) than a compiler

Java programs are compiled into a form called Java byte-code. The JVM executes Java byte-code , so Java byte-code can be thought of as the machine language of the JVM. The Java compiler reads Java source code files (suffixed with .java), translates the source code into Java byte-code , and writes the byte-code into Java class files (suffixed with .class). Generally, Java classes are shipped over the Internet as byte-code. These modules are completely platform independent and are interpreted into machine specific and operating system specific calls by the virtual machine on the target platform.

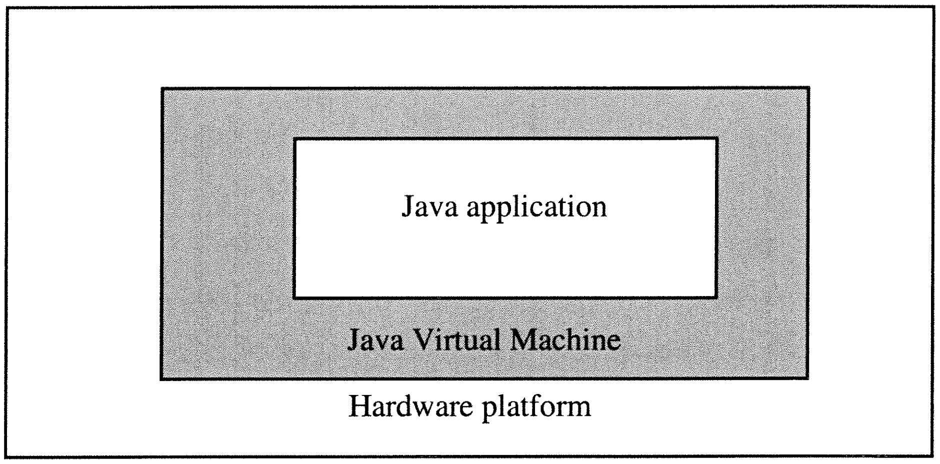


Fig 2.1 Relationship of Java application, JVM, and hardware platform

The JVM plays a central role in making Java portable. It provides a layer of abstraction between the compiled Java program and the underlying hardware platform and operating system, as shown in Fig. 2.1. The JVM is central to Java's portability because Java programs run on the JVM, independent of whatever may be underneath the particular JVM implementation.

JavaSoft, the Sun subsidiary that handles the development of the Java language, controls the specification for what the byte-code must look like, and this specification has remained very stable since the first Java Development Kit (JDK). When the byte-code s are being interpreted by the virtual machine, calls to the standard library functions are handled by the built-in Java libraries, which are also part of the Java specification.

When talking about Java portability, many Java proponents forget to mention the role of Java virtual machine. They gave people a misleading impression that Java is some magic language and is portable just by itself.

The success of Java is due in part to its portability; and paradoxically, this portability arises from the very popularity of the language. Java is so popular and successful that computer vendors have implemented the Java virtual machine on almost all hardware platforms such as Windows, HP UNIX, Sun Solaris, etc. It means that computer vendors have done the platform porting for us. In this way, portability has been achieved.

2.2.2 Reliability

Since Java was initially aimed at simple consumer devices, reliability was a critical objective because in simple consumer devices, software failure is not distinguishable from hardware failure. This goal explains many design choices for Java such as the garbage collection, no explicit pointers, the array boundary checking, the exception and the security management. We will elaborate each choice in a separate subsection.

2.2.2.1 Garbage collection

Java's automatic garbage collection frees Java programmers from spending time worrying about complex and error-prone memory allocation schemes. In Java, there are no memory leaks or dangling pointers - common mistakes under the manual approach.

In C and C++, you create a chunk of memory for an array, for a class, or for any kind of storage needed. You also have to free this memory when you are done with it. But you should not free it if some program actually still wants to reference this memory. In a medium sized application, the separation of memory allocation from de-allocation is not unusual. The worst case is when a third party library is used and you must de-allocate some memory allocated by third party. It is easy to end up with buggy programs that either run out of memory because someone forgot to free allocated memory, or crashes as a program freed memory where a live pointer still points.

Java frees us from the chore of allocating and de-allocating memory with its built-in garbage collection. The Java run-time system can tell (though not 100% accurate) when a block of memory is no longer being used and occasionally collects these up and returns them to the pool of available memory blocks. What we have learned is that garbage collection - while it can have a considerable performance impact - is essential for reliability.

2.2.2.2 References instead of pointers

In a language like C or C++, a pointer is just a simple memory location that holds the address of another memory location. The real problem with pointers is that you can do arithmetic on them and cast them to point to anything you want. Pointers have no checking and no protection associated with them so they could cause unpredictable system damage when manipulated inappropriately. Thus explicit pointers can potentially make a system vulnerable.

In Java, all objects are references (implicit pointer) and you are not allowed to do any arithmetic on them. You can do type casting but the Java compiler and JVM will check the type compatibility of the cast before the reference is used to access an object. If not compatible, either the Java compiler will complain about inconvertible types, or JVM will throw a class cast exception at run-time. Finally, Java does a null pointer check on every reference access. Following null or invalid pointers is a large problem in C or C++. These means contribute to the robustness of a Java program and speed up the development of enterprise applications.

2.2.2.3 Dynamic array boundary checking

Array bound checking is a run-time check to make sure no array reference is out of bounds. This kind of check is common in high-level languages. But system languages like C or C++, for the purpose of convenience and efficiency, reduce array access to pointer arithmetic and do not do any check about it. Dynamic array boundary checking is another measure to ensure the reliability of a Java application.

2.2.2.4 Exceptions

Java has a simple and effective exception mechanism to handle computing errors. This is really important in real-time applications and exceptions have evolved as the best way to handle unexpected conditions in the code.

An exception may indicate an error or an unacceptable situation. Instead of constantly checking all possible variable values in each nested method call and passing error codes up a heavily nested function call chain, exceptions are thrown where a problem is detected. Then, anywhere back up the chain of nested function calls this exception can be caught where it is most reasonable to handle it. Further, Java requires that programmers explicitly handle all exceptions that could be thrown so that with a little more up-front work, the code when compiled will be more robust and require less debugging time.

2.2.2.5 Security

Security is increasingly important as more and more users and their companies are connected to unknown and potentially malicious individuals on a global network.

Java byte-code verifier is a critical security component of the Java VM runtime. It ensures that compiled Java code is formatted correctly and follows the rules that enforce "good behavior". During this process, the verifier applies a theorem-proving algorithm to prove that there is no violation of access restrictions.

The Java class loader, part of Java virtual machine, does not allow Java byte-code to change any system functions. Unlike in C/C++, you are allowed to replace system functions with your own so that you can change their behavior for benign purposes such as debugging very difficult bugs, however it can compromise security.

Finally, Java has a portion of the runtime VM called the security manager. This component is very particular about what kind of I/O functions can and cannot be performed. At run-time, it enforces strict rules about which kinds of these functions will be allowed to prevent uncontrolled I/O calls to write to disk, memory, the video display memory, or even the network, which would constitute a serious breach of security.

With the above three means, the byte-code verifier, the class loader, and the security manager, Java can provide a very secure environment for Java applications.

2.2.3 The Java library

Java has a huge core API library together with some APIs described as Standard Extensions. Java application developers can find most of often used functionality such as data structure support, GUI support, networking, multithreading, Java Database Connectivity (JDBC), etc. The presence of this built-in toolkit speeds up the application development and is also an important reason for Java's success.

Java has a quite complete data structure library and it has built-in classes or interfaces to support Array, Stack, Queue, Linked List, Tree, Hashtable, set, Map, and Vector. Since Java has no explicit pointers, it is easier to understand and to implement data structures in Java than in its counterparts like C++. All these data structure concepts can be applied in any programming language, therefore Java can provide a good introduction to data structures in general and let learners easily have some hands-on practice.

Java supports GUI development through the Abstract Windowing Toolkit (AWT) and the Swing GUI toolkit. The AWT is the Java equivalent of the Microsoft Windows Common Control Library or a Motif widget toolkit. It includes support for simple graphics programming as well as a number of pre-constructed components such as button, menu, list, and checkbox classes so that Java developers can quickly build Java GUI applications. Java 1.2 incorporates the Swing GUI toolkit as a standard Java package with a huge range of new components and controls, and customizable "look-and-feel". Though Swing may be the way of the future for Java GUI developers, it has a steep learning curve because of the complexity of the Swing toolkit.

Java supports network programming by providing classes that can deal directly with sockets, and classes to parse network data and to deal with data representation differences between different hardware platforms. Unlike network programming in C, you need not to handle every detail by yourself. It is more reliable and faster to develop networking programs in Java than in C or C++.

2.3 Weaknesses of Java

Language design is always a compromise between conflicting elements. For example in Java, in order to gain the portability and security, we have to introduce the Java Virtual Machine and therefore sacrifice some program performance. In this chapter, we will look into Java's weaknesses, concentrating on the areas that we plan to correct in this thesis.

2.3.1 Poor readability

Java programs are difficult to read for two seemingly contradictory reasons: sometimes the syntax is cryptic and in other cases it is wordy. In Java, with new concepts such as "pure object orientation", there seems to be little concern with simplicity of expression. As a result, Java programs tend to be overly wordy and algorithm descriptions in Java are much less readable than those in other languages or in pseudo-code. This stands in contrast to previous developments where new languages allowed clearer expression.

Java is based on C, which is hardware oriented and a replacement for assembler language. In contrast, other high level languages such as Fortran, ALGOL, Pascal, etc., tried to be problem oriented -- they are also known as algorithmic languages. When designing the syntax, we would like to keep two rules in our mind: a) prefer algorithmic syntax to hardware syntax such as that of C; b) reduce the number of tokens that must be written.

First, within a Java class, the access modifier (*private*, *protected*, *public*, etc.) applies to each field or method. For example,

```
class JavaClass{
      private int num;
      private float price;
      public int getNum(){return num;}
      public float getPrice(){return price;}
}
```

Other languages such as C++, group class members according to access modifier and do not need to repeat the modifier, i.e.:

```
class CplusClass{
		int num;
		float price;
	public:
		int getNum(){return num;}
		float getPrice(){return price;}
	}
```

Second, the *static* modifier is also applied to class members on individual basis and Java programmers can disperse the static class members all over a Java class. The disadvantages are not only the syntax verbosity but also that, with such a mixed style, developers can not distinguish static members from non-static ones at a glance. Developers can easily make mistakes when programming static methods because a Java static method can call only other static methods or reference only static fields. Actually many professional developers develop their own style to group *static* class members and *non-static* ones. We could extend the use of C++ style and apply it to *static* methods. It is more concise and conceptually cleaner for programmers to write:

```
class A{
	//dynamic part
	int nonStaticMember;
static:
	//static part
	int staticMember;
main(args){}
}
```

This example also shows our proposal for the *main* method. The most wordy syntax in Java is the definition for the main method (if any) in a class: *public static void main(String[] args).* Its method signature is always the same in Java. Maybe the designers aimed to keep all method definitions consistent. But it is really a pain for developers to type in so much, while it does not introduce any new information to the program. At this point, Java can actually learn from some script languages like Perl or JavaScript and simplify the syntax for the *main* method definition to: *main(args)* and let the Java compiler restore it back to the full method definition. By doing this, Java can reduce typing work while keeping all necessary information for compilation.

As far as readability is concerned, Java is much worse than languages like Pascal. In Java, curly brackets are used to delimit all scopes: class, method, and block. When there are many nested scopes and inner classes in one file, all the boundaries are visually blurred, though appropriate code indentation can alleviate the situation to some extent.

We can use a hybrid format: use a begin/end pair to delimit a class declaration and curly brackets to delimit method and block scopes like:

```
[Access_modifier] class classname
begin
public void method1{
...
}
end
```

The begin/end pair is unique within a class and can distinguish more clearly the scope of a class from the inner scopes of the class, thus making programs more readable.

The Java syntax for the IF statement and the WHILE statement are:

```
IF ( expression ) statement
```

and

```
WHILE ( expression ) statement
```

Compared to the Pascal style:

```
IF expression THEN statement
WHILE expression DO statement
```

Java is less human readable. Note that the Pascal style has no parenthesis surrounding the conditional expression and we save one token for each of them.

2.3.2 Object comparison

In Java, there is no syntactic differentiation between object identity and value equality, and the operator "==" is used for both. For example, in the following code snippet:

```
String str1 = new String("Hello world");
String str2 = new String("Hello world");
int int1 =1, int2 = 2;
if(str1 == str2)
  System.out.println("str1 and str2 are equal");
else
  System.out.println("str1 and str2 are not equal");
if (int1 == int2)
  System.out.println("int1 and int2 are equal");
else
  System.out.println("int1 and int2 are not equal");
          ...
```

One may be surprised to know that *str1* is not equal to *str2*, especially to the Java beginner. The confusion stems from the fact that when applied to primitive data types, operator "==" tests for value equality; when applied to objects, it tests for identity equality, i.e., whether two variables point to the same object instance. If we want to test some kind of content equality for two objects, we must either override method *equals()* or program a similar method.

Languages of the Algol family retain the traditional meaning of equality for the "=" operator and use ": =" for assignment. Later when introducing objects, Simula added a distinct operator "==" to test object identity. Simula distinguishes these two equalities by using different operators for each of them. Simula uses operator "=" to test content equality for objects (operator ":=" used for assignment) and operator "==" to test the equality of pointers as shown in the following code segment:

```
TEXT str1, str2;
str1 :- COPY("Hello world");
str2 :- COPY("Hello world");
if (str1 == str2) //identity test
     ...
if (str1 = str2) //value equality test
           ...
```

With such a syntax design, identity equality and content equality are both conceptually and syntactically distinguished, and users will not be confused.

2.3.3 Indexing for String and Vector

In Java, classes *String* and *Vector* are two classes that are used very often. They are essentially arrays with some methods that can manipulate the encapsulated internal elements. However, we can not access their elements using normal array element accessing. Java follows C hardware bent with zero based indexing so that the i-*th* element is denoted *x[i-1]*. We choose not to correct this blemish. We can't access the first character of string *str1* with *str1[0]*. Instead we must use function call *str1.charAt(0)* to do this. Similarly, we have to invoke method *elementAt()* to retrieve an object in a vector. Many programmers are used to the indexing of array with operator [] and they regard the way Java uses to access elements of String and Vector inconvenient, for example:

```
Vector v1 = new Vector(10), v2 = new Vector(10) ;
. . . //store elements into v1
for(int i = 0; i < v1.length(); i++)
    v2.addElement(v1.elementAt(i));
```

While in C++, there is also a vector type and it can behave like a normal array as shown in the following code segment:

```
vector<basic_string> v1(10), v2(10);
. . . //store elements into v1
for(int i = 0; i < v1.length(); i++)
    v2[i] = v1[i];
```

The simplicity and familiarity of the C++ style are obvious.

2.3.4 Type casting

Type casting plays an important role in Java's type system. Java is strongly typed but not statically typed, which means that sometimes the type correctness of a Java program is not known at compile time. The main reason why some type incorrect Java programs can pass the Java compiler, is the type casting operation. Like in a C program, you can cast the type of an expression into many other types in Java and the compiler will accept it. Java is type safe only when the type casting is not involved. For example, Java compiler accepts code fragments like:

```
String str;
Object obj = new Object();
Str = (String) obj;
```

But if we try to run it, the Java virtual machine will report a class casting exception.

Because of the above reason, we can actually relax the type system and let the compiler do the type casting automatically whenever possible. For example, when the LHS and RHS of an assignment are of different types, the compiler could cast the RHS into the type of LHS as illustrated in code snippet that follows:

```
String str1 = "Hello world", str2;
Vector v = new Vector();
v.addElement(str1);
str2 = v[0]; //instead of str2 = (String)v.elementAt(0);
```

However, if the type casting occurred within a complicated RHS expression of an assignment statement, it is hard to do automatic type casting and maybe we have to do it manually. For example, in the following Java statement:

```
substr = ((String)v.elementAt(0)).subString(1,3);
```

It is difficult for the compiler to figure out of what type is the first element stored in vector v. Therefore we must be very careful when researching on automatic type casting.

2.3.5 Complicated I/O

Although in package java.io Java 1.3 has 60 classes that are dedicated to input and output for any difficult I/O task, ironically there is no easy way to do simple I/O jobs like reading an integer from the keyboard. In order to do it, we must go through the steps necessary for most difficult I/O tasks and we may need to program more or less like:

```
BufferedReader in = new BufferedReader(
                        new InputStreamReader(System.in));
String oneLine;
oneLine = in.readLine();
i = Integer.parseInt(oneLine);
```

To show what could be done, we give the C++ equivalent:

```
cin >> i;
```

Moreover, this kind of task is so frequent that it is really an everyday inconvenience in the life of Java programmers, especially the novice.

Maybe Java designers intended to enforce the object oriented program paradigm. They do not want a C-style function *scanf()* in Java to break the object oriented principles. However, there are many ways to remedy the inconvenience within the OO paradigm. Another argument of Java designers is that they want to treat all I/Os in a uniform way --no matter how different in difficulty the I/O tasks are, programmers have to spend almost the same effort to achieve them. This rigidity does not sound tenable enough.

2.3.6 Rigid method selection algorithm

In C, the method name is the only clue to do the method lookup in the symbol table and all methods are in one scope, therefore the compiler will complain about the name conflict if there are two method definitions for one name.

Java supports both method overloading and polymorphism, and they provide flexibility and simplicity for software designers and programmers. These advantages are gained at the expense of complicating the method selection algorithm in the Java compiler.

For a Java compiler, there are three possibilities for a method selection: an exact match found, the most specific method found, and lookup failure. If the name and parameter type match those of the method call, then it is an exact match. For the second case, the name matches, the number of parameters is the same, and the types of parameters are compatible, then the compiler will find a most appropriate one according to the relationship between types. Otherwise, it will report a lookup failure.

However, Java sometimes is not flexible enough in method selection. Let us have a look at the following code fragment:

```
class Test {
      int sum;
      public void add(short item){
            sum += item;
      }
      public static void main(String[] args){
            Test t = new Test();
            t.add(73);
      }
}
```

Everything in the program looks fine: method *add()* expects a short integer and 73 is a short integer. Surprisingly, Java compiler does not compile the program and complains as following:

```
Test.java:8: Incompatible type for method. Explicit cast needed to
convert int to short.
    t.accumulate(73);
```

At this point, Java is really conservative and unreasonable, and it can be improved.

In the coming chapter, we will design a new language to overcome some of Java's disadvantages discussed above.

Chapter 3

Language design of J

3.1 Introduction

Language design is a difficult enterprise that must compromise between high-level abstractions and low-level efficiency concerns. Too much innovation may even be counter-productive if it requires intensive retraining of programmers. Finally, language acceptance may depend as much on lucky timing and marketing strategy as on technical excellence. Java is an obvious success example.

Much progress in computer languages has come from trying to hide hardware dependencies: to make application-oriented languages that focus on concepts that reflect the problem at hand and help the programmer code his solution in a simple and readable fashion. For example, FORTRAN introduced algebraic notation and the concept of arrays for mathematical computations. ALGOL introduced block structure, recursive and formal syntax. Pascal introduced structured (goto-less) programming, strong-type checking, and user defined data types. Finally Simula introduced the object-oriented programming.

C, on the other hand, had a different focus. It was designed to replace machine language in the programming of low-level systems programs such as UNIX. Therefore, run-time efficiency and direct manipulation of hardware was of prime importance. Though it adopted the syntax of high level languages, C remained hardware oriented. The language also reflected the idiosyncrasies of its designer who saw it as a personal tool and not as universal language.

C++ brought about some improvements, such as modularity with object-oriented programming and better type checking with function prototypes. Java added to the security with garbage collection, elimination of direct pointer manipulation, and more stringent type checking. Yet, to allow easy learning by C programmers, Java retains several archaic low-level hardware oriented features. For example: one-dimensional arrays where [1] designates the second element, unstructured case statements, use of the equality symbol "=" for assignment, high dependence of the "cast" operator, etc.

The lack of clarity of Java is especially a problem when it comes to the teaching of both programming and algorithms. However, there are so many useful modern features in the language: objects, graphics, parallelism, distributed execution, etc., that Java is the language of choice in many universities. The solution in many cases is to present examples written in a Java-like pseudo-code rather than Java proper.

This thesis aims to design a simplified external form for the Java language while retaining the Java internals to ease the writing and reading of Java programs. This language is called J.

3.2 Design overview

Different language designers have different points of view and this is reflected in their design principles. It is hard to define a complete list of such principles. However, people can easily find a set of guiding principles when doing language design, such as simplicity, efficiency, safety, expressiveness, modality, and compatibility. Excellent discussions about them could be found in [Bg 96] and [Fwh 92]. In the design of the language J, we try to follow those language design principles and use them as our design guideline.

Since J only extends the external forms of Java and keeps all the Java internals, compatibility with the Java byte-code is our foremost important criterion. J compiler should generate JVM byte code so that J module is compatible with other Java "class" files. Second, J programs should be able to import any Java packages. Otherwise with J you can not do much because you have to program everything from scratch and this deviates from its original purpose: study Java and improve Java. Third, we would like J be able to output a pure Java version of the J code so that we can compare J directly with Java about simplicity and readability.
In some cases, J will accept both Java syntax and more structured J syntax to facilitate testing and comparing about J. Thus J can be used as a pseudo-code compiler for a course that uses a Java Text.

Next, we focus on the simplicity of the program. J tries to shorten the program and reduce the number of keywords that must be written. Serving this purpose, J allows keyword factoring or implicit keywords. In common situations, it assumes "default behavior" and relieves the programmer from necessity of explicit specification.

Java inherits many syntax peculiarities from C or C++ like the syntax for constructs if-else and for loop, which are not user friendly and readable. In these situations, J uses instead the Pascal-style "universal" modes of expression since Pascal is regarded as a model of readable programming language. In some cases, in order to structure program text, we have to reduce some freedom of expression so that code is easier to read.

Finally, we would like to mention that the purpose of this thesis is not to design a robust language for general purpose, but to demonstrate how the Java language could be improved. We also want to show by examples how to apply the compiler techniques for modern computer languages and how to use some compiler tools.

3.3. The language design of J

J modifies partially both the syntax and some syntax semantics of Java. Some modifications are almost trivial to implement when one has the complete source of a compiler available plus a compiler-compiler system. Others are more difficult and require non-trivial modifications to the semantic actions of the compiler. Finally, some modifications that seem to be useful initially may, upon further investigation, prove to introduce some fairly complex semantic problems.

3.3.1 Syntax design of J

The complete syntax grammar in JavaCC is listed in appendix A. Here we would like only to highlight our major modifications to Java.

3.3.1.1 Class declarations

The class is a key concept to Java. Everything in Java: object, variable, method, constant, even main program must be specified with or within a CLASS declaration. This is the first Java-specific concept that must be shown to any student. Thus, it is important to simplify class declarations.

In Java, the class declaration does two things. First, it specifies "dynamic" instance attributes and methods that each object of that class will have. Second, with "static" declarations, it specifies methods and global variables that can be accessed on the basis of class. The static and dynamic declarations can be interspersed and the "static" keyword must be repeated for every static declaration.

In J class declaration, we propose to segregate the static and dynamic declarations by using the "static" keyword to mark the end of dynamic declarations and the start of the static ones. Thus the two functions of a class declaration are clearly separated and the attributes of each are better visually differentiated. Moreover, there are some restrictions on static methods like that static method can only access static class fields and call static class methods. This division is very helpful for us to respect all the restrictions when programming a static method.

Everything is a class in Java and an application is thus expressed as a class with a "main" method. The header for that method is always the same:

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

where the only variant is the name given to the parameter.

J uses "main" as a keyword and simplifies the signature of main method to: *main(args)*and the compiler will take care of the other necessary information and complete the method signature. Thus we save lots of typing for this method without losing any information. Furthermore, to make it clear whether a class is written in "J" or in "Java" - as well as to distinguish class and method declarations - we use "begin" and "end" as class delimiters. The above decisions are exemplified by the Java code snippet and its J equivalent in the following table:

J code	class c1
	begin
	   float x;
	   float y;
	static:
	   private int instances **:=** 0;
	   float pi **:=** 3.14 ;
	   main (args){
	     System.out.println("Hello");
	   }
	end

Java code	class c1 {
	   private static int instances = 0;
	   float x;
	   public static void main(String[] args){
	       System.out.println("Hello");
	   }
	   static float Pi = 3.14 ;
	   float y;
	}

3.3.1.2 If and while statements

For the IF and WHILE statements in J, we prefer the more readable Pascal forms with the keywords THEN and DO instead of the parentheses:

J code	**if a > b then**
	     max := a;
	else
	     max ***:= c;***
	while a > 0 do
	{
	   b ++ ;
	   a := a/2;
	}

Java code	**if (a > b)**
	   max = a;
	else
	   max = c;
	while (a > 0)
	{
	   b ++ ;
	   a = a/2;
	}

3.3.1.3 Use of traditional operators

In mathematical notation, equality is expressed by "=". Languages of the ALGOL/Pascal family follow this convention but other programming languages sometimes use "=" to denote value assignment. With a simple language like Basic, context can distinguish between the two meanings of "=". In other languages, as with C and Java, "=" is an operator that can be used inside more complex *expressions* and another symbol must be used for equality: "==" in the case of Java and C. This works for simple values (like int). For example:

```
int i1 = 99, i2 = 99;
System.out.println(i1 == i2);   // prints "true"
```

With objects, it is tempting to do the same thing and use "==" to test for object equality. With strings, this sometimes works:

```
String s1 = "allo";
String s2 = "allo";
System.out.println(s1==s2);    // prints "true"
```

However, the following example, where both s1 and s2 contain "Hello world" gives a different result:

```
String s1 = new String("Hello world");
String s2 = new String("Hello world");
System.out.println(s1==s2);    // false
```

This is because s1 and s2 are reference variables and the test for equality is done not on the contents of the two objects pointed by s1 and s2 but between the pointer values contained in s1 and s2. In other words, it tests whether s1 and s2 point to the same object: identity testing; not objects of identical value. To test for object equality, one must use the "equals" method declared in the root class (and which must be redefined for every new class). The example below shows the differences:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
System.out.println( s1==s2 );          // false
System.out.println( s1.equals(s2) );   // true
```

Simula has an elegant way to distinguish these operators. It uses operator ":=" for assignment, "=" for value comparison, and "== " for identity comparison.

In J we propose to use the Simula operators: ":=", "=", and "==". For the NOT_Equal operator, we propose to use the C not symbol "!" giving "!=" for not equal and " ! ==" for not identical. Following is an example using these operators:

J code	`String s1 := new String("Hello");`
	`String s2 := new String("Hello");`
	`System.out.println(`**`s1 == s2`**`);      //false`
	`System.out.println`**`(s1 = s2);`**`       // true`

Java code	`String s1 = new String("Hello");`
	`String s2 = new String("Hello");`
	`System.out.println(`**`s1 == s2`**`);      // false`
	`System.out.println(`**`s1.equals(s2)`**`);  // true`

3.3.2 Semantics design of J

When designing a programming language, some designers tend to keep its paradigm as pure as possible, especially for an object oriented programming language. Java claims itself as a pure object-oriented programming language. It is true at a higher level in the sense that all of Java programs consist of classes. However, at a lower level, there are still variables of primitive types and there are many primitive operators operating on them. They are not OOP style and no program can work without it. Therefore the argument on the purity of program paradigm does not make much sense given that we can combine different paradigms in a reasonable way. Like C++, it is a good combination of OOP paradigm and imperative paradigm and it achieves great commercial and academic success. This is one of our guiding principles when designing the semantics of J.

Java does not support explicit operator overloading since there is no language syntax for us to do so in [Ly97]. [Thim 99] pointed out that Java is hypocritical and it supports overloading only when it suits Java itself. The operator "+" is overloaded by Java and it is used very extensively in Java programs. The method *toString* of a class serves this purpose. When + is applied to two String objects, or one String object and one other object (including variable or constant of primitive type), it will invoke the method *toString* automatically. Since Java has overloaded operator +, why not some others if they can improve the expressiveness and the simplicity of a program. In J, we will overload operator =, >, <, >=, and <= whenever needed.

3.3.2.1 Order comparison of objects

Often it is useful to compare two objects to see which one is larger. This is often used in search trees and priority queues.

To this end, Java 2 introduces the Comparable interface which requires classes to implement method *compareTo()* where:

```
a.compareTo(b)
```

returns a value equal to 0, less than 0 or greater than 0 depending on whether ***a*** is equal to, less than, or greater than ***b*** respectively. These conventions give a firm footing to generic algorithms that need to compare objects; however, it leads to wordy and opaque code quite different from the same algorithms when simple values like integers are involved. For the sake of easy reading and writing of code, it would be highly preferable to allow the use of the usual comparison operators, >=, < and <= on "Comparable" objects and this is what we chose for J. Examples are given below:

J code	`if` **`pivot < b.element`** `then`
	`    ...`
	`else if` **`pivot > b.element`** `then`
	`    ...`
	`else`
	`    ...`
Java code	`if (`**`pivot.compareTo(o2.element) < 0`**`)`
	`    ...`
	`else if (`**`pivot.compareTo(o2.element) > 0`**`)`
	`    ...`
	`else`
	`    ...`

3.3.2.2 Indexing of String and Vector

To overcome the problems discussed in previous chapters, we chose to implement the indexing operator [] for String and Vector. The following example demonstrates how the indexing of String and Vector can simplify the writing of programs:

J code	String str := "Indexing";
	char ch;
	int i;
	for(i=0;i<str.length();i++)
	    ch := str[i];
	Vector v := new Vector();
	v.add(str);
	str := (String)v[0];

Java code	String str = "Indexing";
	char ch;
	int i;
	for(i=0; i<str.length(); i++)
	    ch = str.charAt[i];
	Vector v = new Vector();
	v.add(str);
	str = (String)**v.elementAt(0)**;

3.3.2.3 Automatic type casting

Some languages like Algol 68 and Pascal insist that almost all conversion be done through the explicit use of a function. At the other end of the scale, PL/I converts data type as necessary automatically. These implicit conversions are criticized for the fact that some conversion errors are accepted as a reasonable conversion by the compiler. The advantage of Algol 68 and Pascal kind of approach is that the programmer is made aware of almost all conversions.

In J, we made a compromise between both. For some obvious and simple type castings, we use the implicit conversion policy. But for other less obvious conversions, J must do it explicitly. We aim to automate type casting in three areas: 1) arithmetic expression; 2) automate some type cast from a class Object to a more specific Java class used with Collections and utility classes like Vector and Hashtable; 3) method selection.

For the first two cases, it is simpler. When the LHS and RHS of an assignment are of different types, the compiler will cast the RHS into the type of LHS. When writing J code, if this default action does not work, we must do the type casting explicitly. Following is an example of automatic type casting in J:

J code	`String str1 := "Hello", str2;`
	`Vector v := new Vector();`
	`v.add(str1);`
	`str2 :=` **`v[0];`**
	`float f :=` *`0.9;`*

Java code	`String str1 = "Hello", str2;`
	`Vector v = new Vector();`
	`v.add(str1);`
	`str2 =` **`(String)v.elementAt(0);`**
	`float f` **`= (float) 0.9;`**

However, we do not try to automate the type conversion when the type derivation for an expression is complicated, for example,

```
    Vector v = new Vector();
                        ...
    String str := ((String)v.elementAt(0)).substring(1);
```

Because in some complicated cases, it is very difficult to derive (not just guess) the correct target type from the expressions themselves, for example, in the expression `v.elementAt(0).substring(1)`, we can not know easily where we should do type casting and what kind of type casting, only the programmer can know those readily so that programmers should do the type casting explicitly.

3.3.2.4 The Method selection

In Java, there are two primitive type chains: integral types and float types. We can convert one type to the other as indicated by the arrows in the following picture. All other type conversions, like from int to short, are illegal.

$$\begin{tikzcd}
\text{byte} \arrow[r] & \text{short} \arrow[r] & \text{int} \arrow[r] & \text{long} & \text{float} \arrow[r] & \text{double} \\
& \text{char} \arrow[ur] & & & &
\end{tikzcd}$$

We need to introduce the concept of the *distance* between two types, say from A to B. In the above type conversion pictures, if we can reach B directly from A following an arrow, then *distance*(A, B) = 1; if A = B, then *distance*(A, B) = 0; if there is no way from A to B, then *distance*(A, B) = $-\infty$; and the type distance obeys the cumulative rule: *distance(A,C) = distance(A, B) + distance(B, C).*

Next we would like to introduce the concept of the *specificness* of the method definition *methodName*(f_1, f_2, ...f_n), to the method call *methodName*(p_1, p_2, ...p_n), where f_i is the i-th formal parameter type of the method definition and p_i the i-th actual parameter type of the method call:

specificness(*methodName*(f_1, f_2, ...f_n), *methodName*(p_1, p_2, ...p_n)) = Σdistance(f_i, p_i)

The compiler will use this criterion to choose the most appropriate method definition among the candidate method definitions for a method call. Let us have a look at the following code fragment:

```
class Test {
      long sum;
      public void accumulate(int item) {
            sum += item;
      }
      public void accumulate(long item) {
            sum += item;
      }
      public static void main(String[] args){
            Test t = new Test();
            short s = 73;
            t.accumulate(s);
      }
}
```

For the method call *t.accumulate(s)*, the compiler will find that its name and number of parameters are the same as the two candidate methods in class Test, and the type of parameter s, *short*, is compatible with both *int* and *long*. Then the compiler will calculate the specificness of the two candidates. Since *short* is closer to *int* than to *long*, the method: *public void accumulate(int item)* is more specific and will be selected.

After choosing the method definition, the compiler may need to cast the parameters into appropriate types. For the above example, the expression *t.accumulate(s)* will be cast into *t.accumulate((int)s)* implicitly.

In J, we would like to relax the rules for method selection and the type conversion rules for primitive types. We combine Java's two primitive type chains into one and make them compatible in either direction as:

$$\begin{tikzcd}
\text{byte} \arrow[r, leftrightarrow] & \text{short} \arrow[r, leftrightarrow] & \text{int} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \text{long} \arrow[r, leftrightarrow] & \text{float} \arrow[r, leftrightarrow] & \text{double} \\
 & & \text{char} & & &
\end{tikzcd}$$

For a method definition like *void add(int item)*, we still consider method call *add(1.23)* legal. The compiler will cast float literal 1.23 into an integer. One may argue that casting a float into an integer is not well defined for many platforms. That is true. But its behavior can be anticipated in some cases if people can keep the type rules in mind. It is a good experiment to study the type checking.

3.3.3 Some comments

As you may have noticed, we do not handle two weaknesses of Java that we have mentioned in chapter 2: the zero-based array and the Java I/O. Though the fact that *arr[1]* denotes the second element of array *arr* contrasts to our intuition, it is not easy to fix the problem because the implementation of many Java classes like Vector, String, etc., is based on it. If we want to fix the problem, we have to rewrite all those classes.

If simple I/O is available, it will make huge difference for beginner's programs. But it requires a lot of simple-minded coding and the Java working group is preparing a new package, *java.nio*, to enhance the I/O capability of Java. Furthermore, in my opinion, it is more a subject of development of language utilities than a real research issue of the language design itself. Therefore, we choose not to treat the I/O problem.

In next chapter, we will introduce some compiler tools in order to implement a compiler for language J.

Chapter 4

Survey of compiler construction tools

The typical construction of a compiler basically consists of 4 major steps according to [Muc97] and [Mak91]: 1) Lexical analysis to tokenize the input program; 2) Program parsing and production of an intermediate-level representation, usually plus a symbol table; 3) Type checking the intermediate-level representation to check the correctness of the input program; and 4) Generation of code in the target language out of the intermediate-level representation with the help of the symbol table. In addition, a practical compiler always carries out many kinds of optimizations.

Nowadays there are many compiler writing tools that are useful for both research and software production. When people construct a compiler for a language, they rarely build their own lexer and parser from scratch. What they usually do is to use some tool called compiler compiler or parser generator to build the front-end of the compiler. A parser generator is a tool that reads a grammar specification and generates a parser to recognize matches to the grammar and to build a syntax tree as well.

The first compiler compiler dates from July 1960 by Brooker & Morris [Bm62] from Manchester University. Lex & Yacc were the first widely used compiler tools for C applications in Unix environment. Now they are almost replaced by Flex & Bison, which are more powerful and easier to use. Given a grammar description and semantic actions, they will generate a lexical analyzer and parser. The generated program must be fleshed out with more code to do useful work. Flex/Yacc produce C or C++ code.

A more recent compiler generator based on Java is JavaCC. JavaCC has been used for many languages and JavaCC grammars for existing languages are available for C, Ada, XML, SQL, Scheme, and Java itself.

A working compiler needs more than a lexer and parser. It needs at least a structured symbol table to do type checking, a machine code generator, and a convenient interface to the existing code libraries.

If the source code of a compiler for a similar language already exists, it will be easiest for computer language experimentation. In the case of Java, we found a package called Espresso. Espresso is an open-source complete Java compiler for Java 1.0.2 developed at Boston University by Santiago M. Pericas and Karl Doerig. Its main purpose is to serve as a workbench for students interested in applying their theoretical knowledge from programming languages and compiler design and construction for a modern programming language. Espresso uses JavaCC to which it adds its own symbol table. For code generation it also relies on a package from Technical University of Berlin: BCEL (Byte Code Engineering Library). BCEL is responsible for generating Java byte-code and interfacing to Java library.

In this chapter, we provide an overview over JavaCC, BCEL, and Espresso.

4.1 JavaCC

JavaCC is currently the most popular parser generator for Java applications ([JavaCC]). According to Sun Microsystems and Metamata, it has had hundreds of thousands of downloads. JavaCC was developed by Sriram Sankar, Sreenivasa Viswanadha, Rob Duncan, and Juei Chang in Sun Microsystems in 1996, but it is now maintained by Metamata and the main reference is the home page of JavaCC: http://www.metamata.com/JavaCC/. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as building abstract syntax tree (AST), debugging, etc.

The work of using JavaCC is mainly to write the JavaCC grammar, which is a mixture of regular expressions and Java code for semantic actions. The most common output of a parser is an AST. Therefore, before writing the JavaCC grammar file, we must first design a class hierarchy of the AST for the target language. Then we can assign an appropriate AST node to each non-terminal production in the grammar.

4.1.1 JavaCC tutorial by a simple example

We adapt a calculator example that comes with the downloaded JavaCC package to illustrate some features and the use of JavaCC. This calculator accepts an arithmetic expression using only the plus (+) operator and multiply (*) operator operating on integers. For example,

```
? 12 + 23
= 35
```

This example will be reused in the following two sections.

When writing JavaCC grammar for this example, we assume a hierarchy of Java classes as depicted below. These classes will be used in semantic actions associated to the grammar rules.

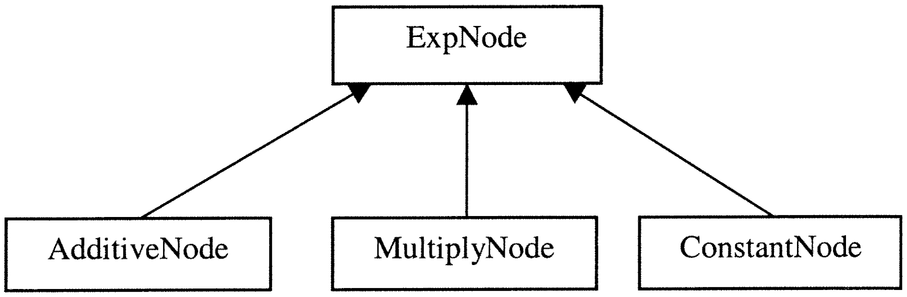


Fig. 4.1 Class hierarchy for JavaCC example

Class ExpNode is the base class and all others are subclasses of class ExpNode. We do not introduce much about the classes except for their constructors, as the class names are quite self-explanatory. The constructors for them are listed below since they will be used later in the JavaCC grammar:

```
public AdditiveNode(ExpNode left, ExpNode right);
public MultiplyNode(ExpNode left, ExpNode right);
public ConstantNode(String numbers);
```

Now we have all the elements to write the JavaCC grammar for our simple calculator. As usual we list the grammar in Courier font. The grammar starts with a Java "main" method and this is followed by the grammar proper. Note that the words in capitals (like LOOKAHEAD) are keywords for JavaCC and the line numbers are not part of input and they are used to facilitate the explanation.

```
1.  options {
2.  LOOKAHEAD=1;
3.  }
4.  PARSER_BEGIN(Calculator)
5.  public class Calculator {
6.      public static void main(String args[])
```

```
                       throws ParseException {
        Calculator parser = new Calculator(System.in);
        ExpNode ast;
        while (true) {
            System.out.print("Enter Expression: ");
            System.out.flush();
            try {
                ast = parser.expression();
            } catch (ParseException x) {
                System.out.println("Exiting.");
                throw x;
            }
            //do something with the AST, for example,
            //evaluating and displaying result
        }
    }
}
PARSER_END(Calculator)

SKIP :
{
    " "
    |      "\r"
    |      "\t"
}

TOKEN :
{
    < EOL: "\n" >
}

TOKEN : /* OPERATORS */
{
    < PLUS: "+" >
    | < MULTIPLY: "*" >
}

TOKEN : /* Integer Constants */
{
    < CONSTANT: <INTEGER> >
    | < #INTEGER: ( <DIGIT> )+ >
    | < #DIGIT: ["0" - "9"] >
}

ExpNode expression() :
{
    ExpNode node;
}
{
    node = sum() <EOL>
      { return node; }
    | <EOL>
      { return null; }
    | <EOF>
      { return null; }
}

ExpNode sum() :
{
    ExpNode node1, node2;
}
{
    node1=term()
    ( <PLUS>
      node2=sum()
    {node1 = new AdditiveNode(node1, node2);}
    )*
    { return node1;}
}

ExpNode term() :
{
    ExpNode node1, node2;
}
{
    node1=element()
    ( <MULTIPLY>
      node2=term()
      { node1 = new MultiplyNode(node1, node2);}
    )*
    { return node1;}
}

ExpNode element() :
{
    ExpNode node1;
    Token t;
}
{
    t=<CONSTANT> {node1 = new ConstantNode(t.image);}
    {return node1;}
}
```

From the above example, we can see that the JavaCC grammar file looks like a Java program. It starts with an option block (which is optional) to finely tune the efficiency of the resulting parser or to turn on some debugging feature, etc. In the example, we have only the instruction "LOOKAHEAD = 1", which tells the parser to look ahead only one token.

Then follows a Java compilation unit enclosed between "PARSER_BEGIN" and "PARSER_END", which is used to specify the Java Parser class name, in our example, Calculator. The code between them will be copied to the resulting Java parser class.

Next is a skip block to tell the parser which character to be ignored when parsing a program. Meanwhile, they act as token delimiters. In our case, character blank, carriage return, and tab will be ignored.

Following is a list of token definitions. A token is a sequence of character that matches a pattern defined by a regular expression. In the example, we defined four tokens: EOL, PLUS, MULTIPLY, and CONSTANT. Whenever a token is recognized, an object named Token will be returned to parser. Note that Token is a built-in class of JavaCC and it has a field named *image* containing the token string.

The last and most important section is the JavaCC productions part. In JavaCC grammar, the syntax of a production is:

```
<Java_type> <Java_identifier>"("<Java_parameter_list>")" ":"
<Java_block>
"{"
        <expansion_choices>
"}"
```

The *Java_identifier* is the non-terminal name and has a method-like signature. It has a return type and parameters (if necessary). Actually each non-terminal is translated into a method of the same signature (including name, return value, and parameters) in the generated parser class. Parameters and return value of the non-terminal are the means to pass values up and down the parse tree. If a non-terminal includes some other non-terminals, they appear in the production just like any method. A non-terminal can also include itself. Therefore JavaCC production can be recursive. For example, in non-terminal *sum()* at line 58, it may contain one *term* and may call itself zero or more times

```
term() ( <PLUS>  sum())*
```

The <J*ava_block> in a JavaCC production* is a block of Java declarations and statements. They are visible to the embedded Java code in semantic actions in the production. JavaCC does not do any processing on this block, it simply copies them to the beginning of the resulting method. Every time this non-terminal is involved in the parsing process, this Java block will be executed.

The <expansion_choices> is a list of possible choices separated by the OR character "|", while each choice could be a concatenation of non-terminals and tokens. In the example, the non-terminal *expression*() at line 46 has three expansion choices:

```
sum() <EOL>| <EOL> | <EOF>
```

which means that a legal expression non-terminal must be only one of the three choices. Now we can go over each production in our simple example.

The first and root non-terminal is *expression*(). It returns an *ExpNode* object and has no parameter. In the Java block, it declares an *ExpNode* object, *node*. After each expansion choice, there is an optional Java code block called *parser actions* or *semantic actions*. It will be executed if a match for that expansion is recognized. It is copied into the Java method parsing the non-terminal at the appropriate location. This is a new concept introduced in JavaCC. If an expression consists of a non-terminal *sum()* ending with an EOL character. The return result of non-terminal *sum ( )* will be assigned to *node* and be returned to parser as the internal representation of the calculator expression. If only EOL or EOF is recognized, a null pointer will be returned.

The second non-terminal is *sum*(). One point special in it is that this non-terminal production is recursive and has a regular operator *. This operator means the sequence "*<PLUS> sum*()" can appear zero or more times. The Java block inside the * operation will be executed and how many times it will be executed depends on how many times the sequence "*<PLUS> sum*()" is matched.

The top-down decomposition will continue through non-terminal *term ( )* to non-terminal *element*(). The expansion choice is token *<CONSTANT>*. When there is a match for it, JavaCC creates an object of type Token. You use the object in your semantic actions. Note that we do not need an extra lexer when using JavaCC.

At this point we have described the skeleton of a JavaCC grammar file.

4.1.2 JavaCC grammar for Java

There are several JavaCC grammar files coming with JavaCC package for various Java versions. Since our work is on Java 1.0.2, we will introduce briefly the grammar file for this version.

All reserved words, primitive operators, and literal of Java are defined as tokens at the beginning. Then the grammar starts with the top non-terminal production *CompilationUnit*, which represents a Java file to be compiled. It may include three components: optional package declaration, optional import declaration, and optional type declarations. Usually a Java file has at least one type declaration, either an interface or a class. The top-down decomposition will continue through Java class field and method declaration, through all kinds of Java statements, through all kinds of Java expressions, until it hits the token as end points.

The complete JavaCC grammar file for Java 1.0.2 has 983 lines and altogether 79 productions to describe the Java language, out of which 25 are for *expressions* and 19 are for *statements*. At first sight, it is overwhelming. However, due to its top-down structure, it is easy to read and understand.

Note that it is only the grammar to define a legal Java program. Usually we should change or embed some Java code as semantic actions into the grammar file according to our needs. For example, each non-terminal production's return type was originally void and we must change it to return some kind of syntax tree node in order to build an AST.

4.1.3 Advantages of JavaCC

The most widely used compiler compiler tool is Lex/Yacc ([Lmb95]). Compared to them, JavaCC has the following advantages:

JavaCC integrates the functionality of both Lex and Yacc in that you can specify both the lexical rules (regular expression) as well as parsing rules (BNF) in the same input file. Therefore, with JavaCC we need fewer steps to construct a compiler.

JavaCC generated parser and lexer are self-contained in that all code required to run the parser is generated and no support libraries are required.

JavaCC grammar is more readable than the grammar for Lex/Yacc. At the right-hand side of each production, JavaCC can mix the non-terminals with Java code (semantic actions), while with Lex/Yacc actions appear only at the end of production. Actually, within many productions certain actions are only related to part of the production. If we can insert actions as soon as we see that part, the grammar file will be more readable and easier to debug.

JavaCC puts less restriction on language grammars. Lex/Yacc can handle only the LRLA(1) grammar. Though LRLA(1) is the standard of syntax design for most existing programming languages, in some case we want to design our own language or to modify an existing language. Then we will find that LRLA(1) is very restrictive. JavaCC accepts LL(k) grammars, where LL(k) means the look-ahead number is variable and you can look ahead as far as you want. In practice, we can easily design a language in which most part is LL(1) and there are only a few choice points where we must look ahead more than one token. In this way, we can get an efficient parser with little restriction on grammar.

JavaCC is very flexible when constructing the AST. Sometimes we need to pass information up and down the AST. JavaCC can do this naturally since its non-terminal production can have parameters and return a result like a normal method call. While it is very difficult to go up to a higher level production with Lex/Yacc.

4.2 Byte-code Engineering Library (BCEL)

BCEL ([Bcel98]) is a framework package for Java byte-code engineering. It provides users a convenient way to read, to create, and to manipulate a Java class file.

4.2.1 Java class file format

To understand the importance of BCEL, we need to have an overview on the Java class file format. [Ly97] has a detailed introduction of Java class file. Each Java class file contains a single *ClassFile* structure:

```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
}
```

Note that all the types mentioned in the above declaration are defined by JVM. Below, we explain their meaning:

magic:

The *magic* item supplies the magic number identifying the `class` file format; it has the value `0xCAFEBABE`.

minor_version **and** ***major_version:***

The values of the *minor_version* and *major_version* items are the minor and major version numbers of the compiler producing this `class` file.

constant_pool_count:

It gives the number of entries in the *constant_pool* table of the `class` file.

constant_pool:

The *constant_pool* is a table of variable-length structures representing various string constants, class names, field names, and other constants that are referred to within the *ClassFile* structure and its substructures.

access_flags:

The *access_flags* item is a mask of modifiers used with class and interface declarations.

this_class:

The value of the *this_class* item must be a valid index into the *constant_pool* table. The *constant_pool* entry at that index must be a *CONSTANT_Class_info* structure representing the class or interface defined by current `class` file.

super_class:

The value of the *super_class* item either must be zero or must be a valid index into the *constant_pool* table to indicate the superclass of the current class or interface whenever necessary. In Java, only class *java.lang.Object* has no superclass.

interfaces_count:

The *interfaces_count* item gives the number of direct *superinterfaces* of this class or interface type.

Interfaces:

Each value in the *interfaces* array must be a valid index into the *constant_pool* table to represent an interface that is implemented by current class file.

fields_count:

Item *fields_count* gives the number of *fields* that are defined by this class file.

fields:

`Item` *fields* represents an array of *field_info* structures each of which describes in detail a *field* declared by this class or interface type. It does not include items representing *fields* that are inherited from superclasses or superinterfaces.

methods_count:

Item *methods_count* gives the number of *methods* that are defined by this class file.

`methods:`

Item *methods* is an array of `method_info` structures each of which gives a complete description of Java Virtual Machine code for a method in the class or interface.

From the brief description, we can see that the structure of a class file is very complex. If we are asked to produce such a structure from scratch manually, it is very difficult to get all the details right and to ensure the integrity and validity of a Java class file. The package BCEL can bail us out of this hard situation so that we can concentrate better on high-level language concepts. For example, if we want to remove or add an instruction to a Java method directly, we must calculate the concrete byte-code offset, which is tedious and error-prone. If we use the BCEL package, a simple delete or add method call will handle every detail for us.

BCEL is composed of two main packages: package *classfile* and package *generic*. The package *classfile* contains the classes that describe the structure of a Java class file and a class file parser. It can analyze and give a static view upon Java classes without having the source files at hand. While the package *generic* contains the classes to dynamically create and modify class objects and byte code instructions, and it may be used to optimize Java byte code or to implement the back-end (code generator) of a Java compiler.

4.2.2 Sub-package *classfile* of BCEL

Figure 4.2 shows an UML diagram of the hierarchy of classes of this package. The most important classes in this package are *JavaClass* and *JavaParser*. Objects of JavaClass in most cases are created by a *ClassParser* object which is capable of parsing binary class files and mapping all of the binary components and data structures into all kinds of classes as described in Fig 4.4. A *JavaClass* object basically consists of fields, methods, and symbolic references to the super class and to the implemented interfaces. *JavaClass* package has a *ConstantPool* class to represent a constant pool of a Java class file. *ConstantPool* objects contain an array of fixed size of Constant entries, which may be retrieved via method *getConstant*() which takes an integer index as argument. Class *Methods* and *Fields* represent methods and fields of a class, respectively.

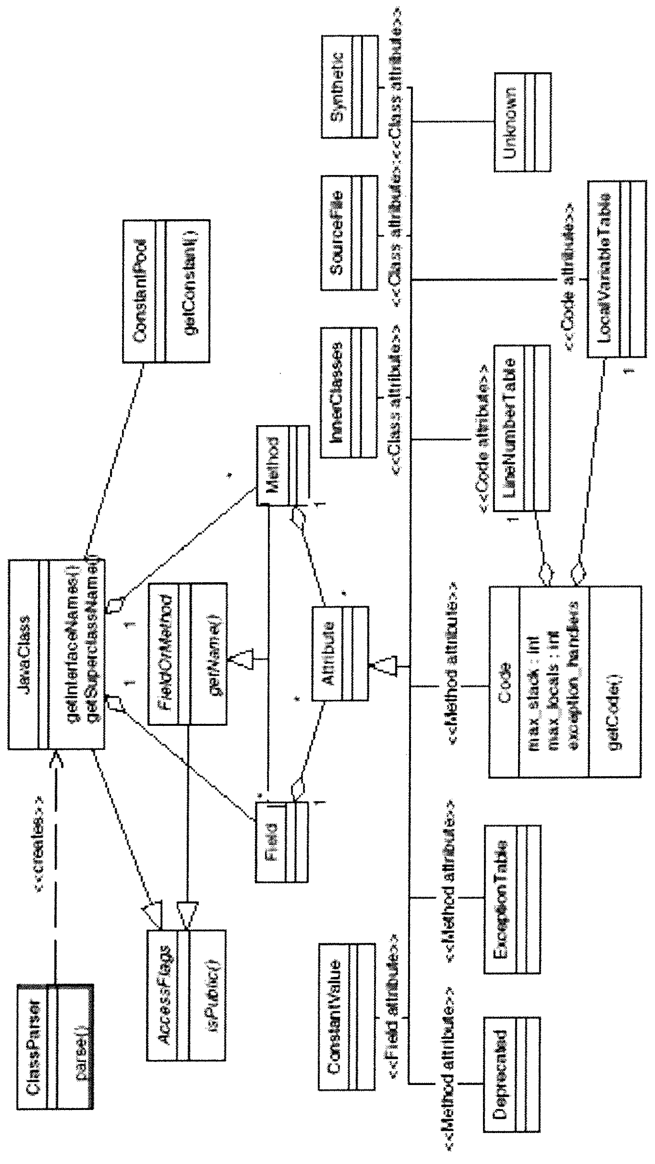


Figure 4.2 UML diagram of package *classfile* in BCEL

The package *classfile* provides a *Repository* class to read class files and to obtain *JavaClass* objects therefore we can interface with existing class files. We need only to specify the name of the class file to the *Repository* and it will analyze the class file and fill all the attributes of the *JavaClass* object, i.e., all the information of a class file is stored in the object for future use. The usage of *Repository* is as simple as:

```
JavaClass myclass = Repository.lookupClass("java.lang.String");
```

Moreover, *Repository* maintains information about class interdependencies, e.g., whether a class is a sub-class of another, or whether a class implements a specific interface, etc. This functionality is very important when we want to do type checking.

After we obtained a *JavaClass* object, information about the class file components may be accessed or modified via the set and get methods. All the class file components like Method, Field, etc., also define a *toString*() which will help to print them out in a readable way so that we can implement a class viewer easily.

4.2.3 Sub-package *generic* of BCEL

Figure 4.3 shows an UML diagram of the hierarchy of classes of package *generic*. It can help compiler writers to generate Java byte code. The concept of generic is essential for code generation. For example, in the JVM, there are many variants of abstract instructions ADD depending on the type, the location and even the value of the operand. In code generation, we want to build internal representations of the program using the abstract nodes like ADD and have the generation routines produce the correct byte codes.

The most important classes are *Classgen*, *ConstantPoolGen*, *MethodGen,* and *FieldGen*. They work together to compile methods and fields of a class to construct a *JavaClass* object. The constant pool generation of a class is handled by class *ConstantPoolGen* that offers methods for adding different types of constants and constructing a constant pool for a Java class. *MethodGen* and *FieldGen* are aimed to construct a *Method* object and a *Field* object, respectively, for a *JavaClass* object.

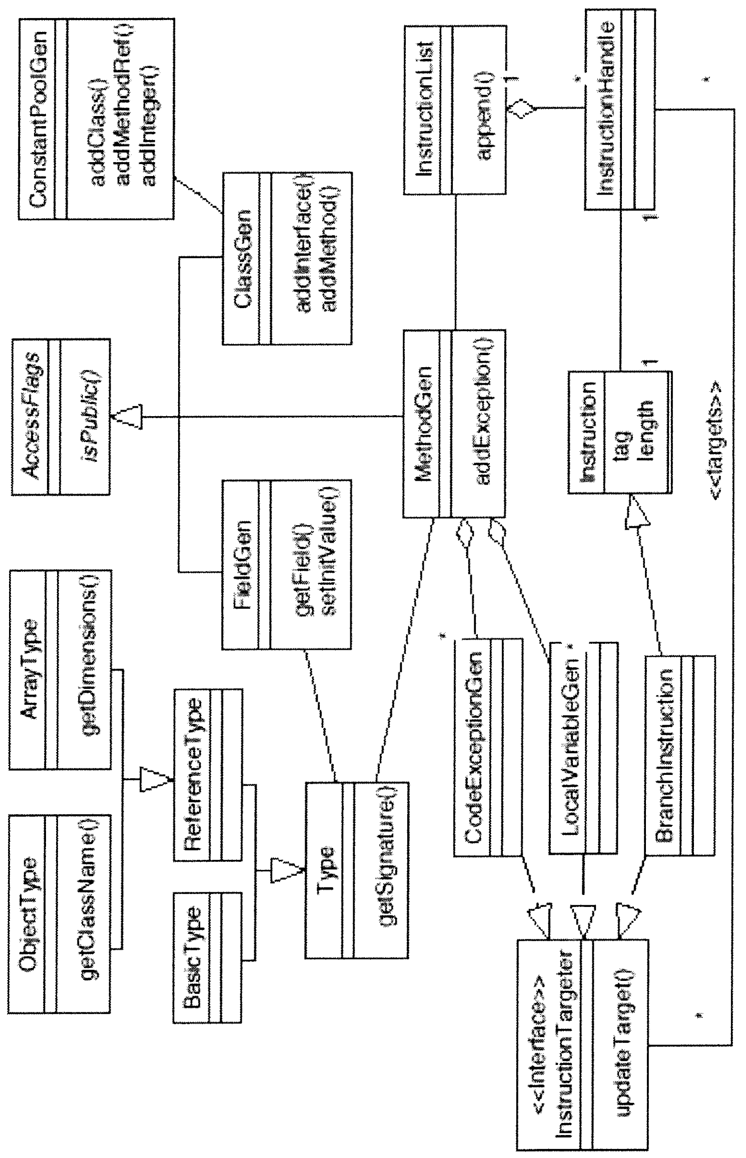


Fig. 4.3 UML diagram of BCEL *generic* package

4.2.3.1 Types

ClassGen abstracts from the concrete details of types by introducing the *Type* class, which is used, for example, by methods to define their return and argument types. Concrete sub-classes are *BasicType*, *ObjectType*, and *ArrayType*. The last one consists of the element type and the number of dimensions. For basic types, *ClassGen* offers some predefined constants. For example, the return type and argument types of the Java main method is represented by:

```
Type return_type = Type.VOID;
Type[] arg_types = new Type[] {new ArrayType(Type.STRING, 1)};
```

Type objects can be converted to textual signatures with *getSignature*().

4.2.3.2 Generic *fields* and *methods*

Fields are represented by *FieldGen* objects. Generic *methods* contain methods to add local variables, exceptions the method may throw, and exception handlers. Exception handlers and local variables contain references to byte code addresses. Generic (non-abstract) methods refer to *instruction lists* that consist of instruction objects. References to byte-code addresses are implemented by handles to instruction objects. This will be explained in more details in the following sections.

4.2.3.3 Instruction list of BCEL

In BCEL, there is a class corresponding to each JVM instruction. Modeling instructions as objects may look somewhat odd at first sight, but in fact it enables a compiler writer to obtain a high-level view upon control flow without handling details like concrete byte code offsets.

Instructions consist of a tag, i.e., an opcode, their length, and an offset (or index) within the byte code. Instructions are grouped via sub-classing, the type hierarchy of instruction classes is illustrated by figure 4.4. The most important family of instructions is the *branch instructions,* like *goto*, that branch to targets somewhere.

An *instruction list* is implemented by a list of *instruction handles* encapsulating instruction objects. References to instructions in the list are thus not implemented by direct pointers to instructions but by pointers to instruction *handles*. This makes *append*, *insert,* and *delete* code segments very simple. Since we use symbolic references, computation of concrete byte code offsets does not need to occur until finalization, i.e., until the user has finished the process of transforming or generating code.

The class *InstructionList* representing an instruction list is one of the most important classes of BCEL. It has the methods to create a new instruction list, append and add an instruction or another instruction list, and delete an instruction from an instruction list anywhere. The following listing is a simple example showing the use of class *InstructionList* and some of its methods:

4.2.3.4 Finalizing a JavaClass object

When the instruction list is constructed, all symbolic references must be mapped to real byte code offsets. This is done by the method *getByte-code* () called by the method *getMethod*() of class *MethodGen. ClassGen* object will call method *getJavaClass*() to obtain a *JavaClass* object, which contains all information about the Java class to be compiled and can dump valid Java byte code to file, i.e., finish the compiling process. Thus we can produce byte code in a controlled manner.

4.2.4 An example using BCEL

We return to the simple example used for the JavaCC tutorial to illustrate how to use the BCEL APIs. Whenever there is an expression recognized, for example 23 + 3, it will imply a Java class Calculator as follows:

```
import java.io.*;

public class Calculator {
      public static void main(String[] args){
            int x;
            x = 23 + 3;
            System.out.println(x);
      }
}
```

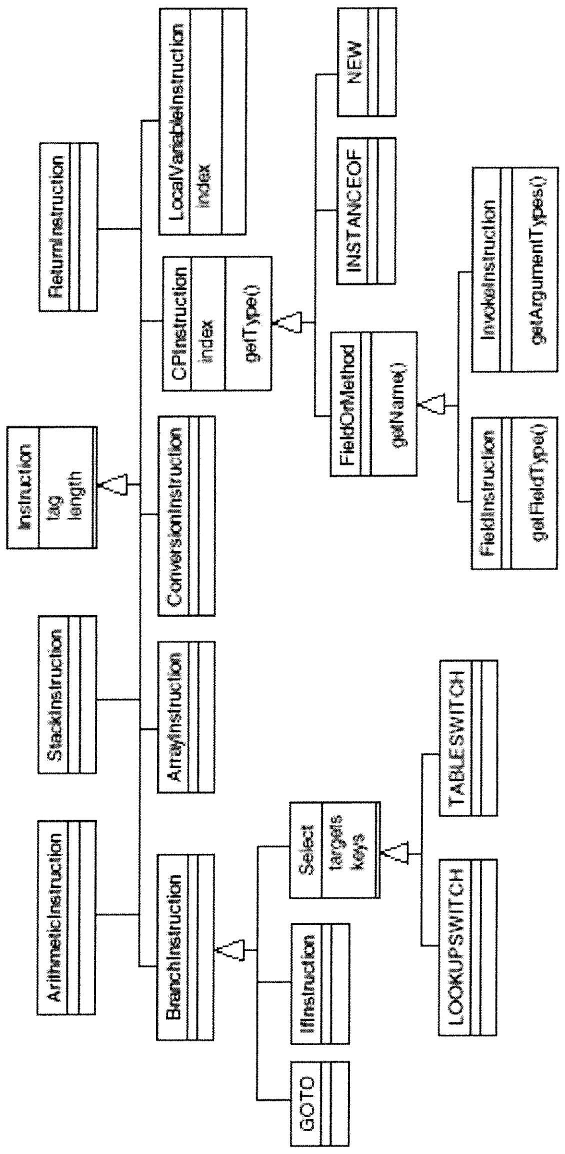


Fig. 4.4 UML diagram for the instruction API

We will use BCEL to produce Java byte code by hand for the complete short program shown above. To do this, you must have a good understanding of Java byte code and the BCEL APIs.

```
import de.fub.byte-code .generic.*;
import de.fub.byte-code .Constants;

public class CalculatorCompiler {
  public static void main(String[] argv) {

       /** create a ClassGen object for class Calculator */
      ClassGen  cg = new ClassGen(
            "Calculator",           // class_name
            "java.lang.Object",     //super_class_name
            "<generated>",          //file_name to dump class file
            Constants.ACC_PUBLIC |
            Constants.ACC_SUPER,    // class access_flags
            null);                  //array of interfaces implemented

       //get the constand pool from the ClassGen object
       ConstantPoolGen cp = cg.getConstantPool();

       //create an instruction list object for the main method
       InstructionList il = new InstructionList();

       /** compile the main method  */
       MethodGen  mg = new MethodGen(
            Constants.ACC_STATIC |
            Constants.ACC_PUBLIC,       // access flags
            Type.VOID,                  // return type
            new Type[]{new ArrayType(Type.STRING, 1)},
                                        // parameter types array
              new String[] { "args" }, //parameter names array
            "main",                     // method name
            "Calculator",               // within which class
             il,                        // instruction list of method
            cp);                        // associated constant pool

       /** Add constants to constant pool
       * and later on they can be referenced by index.
       */
       int system_out = cp.addFieldref("java.lang.System",
             "out", "Ljava/io/PrintStream;");

       //compile "int x"
       LocalVariableGen lg;
       lg = mg.addLocalVariable("x", Type.INT, null, null);

       /* get the index for local variable x for later reference */
       int x = lg.getIndex();

       /* start to add instructions to instruction list*/


       //compiler 23 + 3
       //load int constant 23 onto stack
       il.append(ICONST(23));

       //load int constant 3 onto stack
       il.append(ICONST(3));
```

```
            //add the top two integers in the stack togeter
            il.append(IADD());

            //store the top integer in stack into local variable
            // and  mark the validity beginning of local variable "x"
            lg.setStart(il.append(new ISTORE(x)));

            //load the local variable onto stack
            il.append(ILOAD(x));

            //call method System.out.println(x)
            il.append(new INVOKEVIRTUAL(
                 cp.addMethodref("java.io.PrintStream","println", "(I)V")
                                      )
                    );

           //add a return instruction to the end of il
           il.append(InstructionConstants.RETURN);

           //set the max stack size of JVM
           mg.setMaxStack(2);

           //add the method with its instruction list to the class
           cg.addMethod(mg.getMethod());

           /* Add public <init> method, i.e. empty constructor  */
           cg.addEmptyConstructor(Constants.ACC_PUBLIC);

           //Get JavaClass object and dump it into a class file.
           try {
                cg.getJavaClass().dump("Calculator.class");
           } catch(java.io.IOException e) {
              System.err.println(e);
           }
       }
}
```

4.3 Espresso

Espresso has a class to represent every Java program component, like the statement, the expression, the variable, the literal, etc. It uses JavaCC to tokenize and parse the program, and build an abstract program representation: an AST, according to the given Java grammar file and the semantic action embedded within the grammar as well as a symbol table which has information for all the variables and constants. This is used to handle the package and block structure of Java programs and to do type checking.

Espresso also has a Java import manager to interface between the symbol table and Java "class" files. In particular, the import manager is used to enter into the symbol table information about fundamental classes such as *String*, *Object*, etc.

After Espresso has finished all the traversals of the AST and attached all needed information, it uses the generic package of BCEL to produce Java byte code. In the following sections, we will elaborate on the procedures mentioned above.

4.3.1 Class design of Espresso

The first step of Espresso is to define a hierarchy of Java classes to internally represent every possible program structure. Some of them are listed in the table below:

CompilationUnitNode	TypeDeclarationNode	ClassDeclarationNode
ClassBodyDeclarationNode	ConstructorDeclarationNode	ConstructorInvocationNode
StaticInitializerNode	FieldDeclarationNode	MethodDeclarationNode
InterfaceDeclarationNode	InterfaceMemberDeclarationNode	VariableDeclaratorNode
FormalParameterNode	LocalVarDeclarationNode	ArrayInitializerNode

In addition, there are 16 different *StatementNodes* like *ForStatementNode*, *WhileStatementNode, IfStatementNode*, etc., and 44 different *ExpressionNodes* such as *AndExpNode*, *AssignmentNode*, *AdditiveExpNode*, etc.

The class hierarchy is quite big and all the classes could be categorized into three main categories: *class-level* declaration nodes, *statement* nodes, and *expression* nodes that correspond to Java *declarations*, *statements,* and *expressions,* respectively. Each *CompilationUnitNode* represents a Java class or Java interface to be compiled. The meaning of other classes is evident from the names. All the classes are subclasses of class *SyntaxTreeNode*.

4.3.2 Program parsing and construction of the AST

Given the above classes, Espresso will add semantic code to the JavaCC grammar file. The resulting grammar will be fed to JavaCC to construct a LL(k) parser for Java. The generated Java parser is used to parse the Java source code. First it checks the Java code conforms to the Java grammar and reports errors and stops the compiling process whenever it finds one error, which means that the error-tolerance of Espresso is quite poor and can report only one error at a time. Meanwhile, Espresso constructs an AST to represent the original source code. Each node of this tree is an object of a class described in the previous section. If Espresso verifies that the program contains no grammatical error, it will return an AST as the result of compilation. Later, the type checking and code generation will work on AST.

We reuse the calculator example in previous section to demonstrate the procedure. In order to facilitate the understanding, we repeat the Java code here.

```
import java.io.*;

class Calculator {
      public static void main(String[] args){
            int x ;
            x = 23 + 3;
            System.out.println(x);
      }
}
```

Espresso will build an AST shown in Figure 4.5:

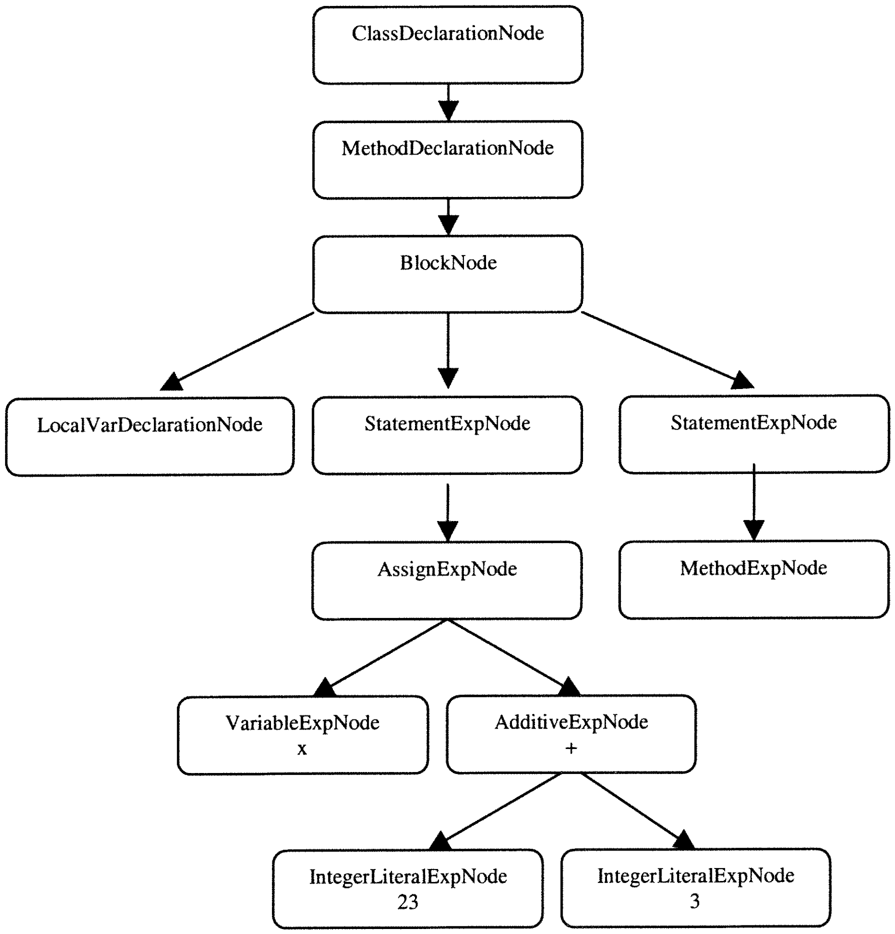


Fig. 4.5 The partial AST for the Calculator example

The AST can not keep all the program information by itself and it works together with symbol table.

4.3.3 Symbol table

With the syntax tree, the symbol table is one of the two major data structures used in the compiler. It is intimately involved with the parser, which may need to enter information directly into the symbol table or to consult it to resolve ambiguities.

The symbol table is mainly a typical dictionary data structure. The principal operations are *insert* and *lookup*. In some other cases, *delete* may also be necessary. Typically, the information which needs to be stored in the symbol table includes data type information, program scope information, and information on eventual location in memory. The first two types of information are used mainly for parsing and semantic analysis like type checking, and the last one is used at the stage of code generation.

In the symbol table, the attributes bound to a name vary with the kind of declaration. For example, a ClassDeclarationNode has a class name whereas *MethodDeclarationNode* has a method name.

Espresso's symbol table is implemented as a flat data structure using a Java Hashtable. In order to resolve imbricate scope with a flat data structure, Espresso uses a naming convention similar to UNIX paths. Another solution would be to use a tree-structured table.

All the entries in a symbol table are fully resolved in order to keep them unique. For example, the name of a field is recorded as *packageName.className.fieldName* and a method as *packageName.className.methodName*. In order to deal with the different scopes within a method, Espresso uses an integer number *scopeid* to encode the variable name in symbol table. The symbol table of Espresso encodes a formal parameter as *packageName.className.methodName.**0**.formalParameterName* and a local variable as *packageName.className.methodName.**scopeid**.localVariableName*. Since the formal parameter is always at the top level of a method, its scopeid is fixed to 0

For the calculator example, a symbol table as shown by the following table is constructed when building the AST.

	Symbol (key)	Item
1	Calculator	class
2	Calculator.main	method
3	Calculator.main.0.args	formal
4	Calculator.main.0.x	Local variable
5	Java.lang.System	Class(JIM)
6	java.io.PrintStream.println.()	Method (JIM)
7	java.lang.System.out	Field (JIM)
8	java.io.PrintStream	Class (JIM)

Given the above encoding schemes, we can guarantee that all the entries in the symbol table are unique and original scopes are preserved.

The imported entity is manipulated the same manner in the symbol table as the user-defined entities present in the compiled unit. It allows for a uniform handling during type checking and code generation.

4.3.4 Java Import Manager (JIM) of Espresso

JIM is an interface to the Java library and existing compiled Java class files. When we encounter a symbol name that is not defined within current compilation unit, we must use JIM to determine whether it is defined in a certain class imported by the source code. Then it makes appropriate portions of compiled Java class files available to the environment of Espresso, specifically, to enter an entry in the symbol table and build an appropriate Node object to represent the precompiled code.

JIM mainly uses class *JavaClass* of BCEL to achieve this goal. First it constructs a *JavaClass* object from an existing Java class file and uses the facilities of *JavaClass* to retrieve class information from the *JavaClass* object and add appropriate entries into the symbol table. In this manner, Espresso can treat an imported class (already compiled) the same way as a user-defined class in the source code.

4.3.5 Type checking

One of the principal tasks of a compiler is the computation and maintenance of information on data types (type inference) and the use of such information to ensure that each part of a program makes sense under the type rules of the given language (type checking). Usually, these two tasks are closely related, performed together, and referred to simply as type checking.

After the *JavaParser* parses Java code and constructs an AST successfully, type checking is performed on the AST with help from symbol table. Each node of the AST provides a type check method that is responsible to perform the type checking of the language construct it represents, and to propagate type checking down to its component nodes in the AST if required.

During parsing, types declared for methods, fields, formal parameters, and local variables are collected by the parser, and turned into instance of classes of the type hierarchy. Constructing type instances also includes encoding these types according to the encoding scheme defined by the JVM to make them comparable to the ones loaded from class files.

In addition to determine the resulting types of expressions, type checker must verify that access restriction for Java packages, classes, methods, and fields are respected.

This section introduces to the Espresso's type checking concepts and highlights some difficult or non-standard problems.

4.3.5.1 Type hierarchy of Espresso

The type hierarchy developed for Espresso is one of the most important and central concepts of Espresso. Almost all non-trivial functions for type checking are integrated in this hierarchy. It also plays a major role in supporting code generation.

Java's type system distinguishes between two kinds of types: primitive types and reference types. Primitive types include all the numeric types, *void* type, and *boolean* type. Numeric types are further divided into integral types (which include the types *byte*, *short*, *char*, *int*, and *long*), and floating point types (*float*, and *double*). There are three kinds of reference types in Java: class types, interface types, and array types. In addition, a special null type is also defined.

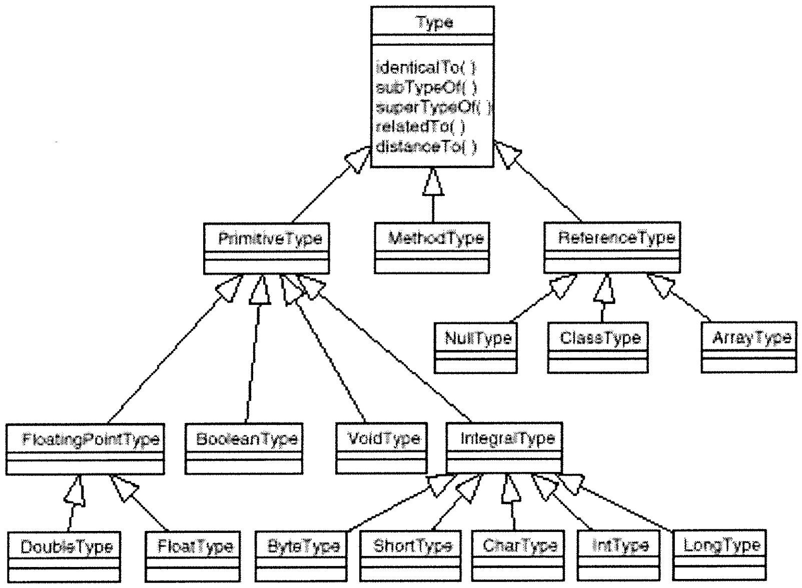


Figure 4.6. UML class diagram of Espresso's type hierarchy
(From the technical report of Espresso)

Espresso's type hierarchy also contains a special type for methods. Although the Java language has no objects of method type, the availability of this special type provides an elegant solution for many problems during type checking and code generation.

4.3.5.2 Type Relationships

Given the type hierarchy, we must somehow find the relationship between two types, which is essential in type checking. All these relationships can mainly be divided into four categories: identical, subtype, supertype and related.

Type distance is invented to quantify the relationship between two types. Table 4.1 defines the distances between all the primitive types.

Table 4.1 the type distance matrix in Espresso

	boolean	byte	short	char	int	long	float	**double**	class
boolean	0	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞
byte	-∞	0	1	-∞	2	3	4	**5**	-∞
short	-∞	-∞	0	-∞	1	2	3	4	-∞
char	-∞	-∞	-∞	0	1	2	3	4	-∞
int	-∞	-∞	-∞	-∞	0	1	2	3	-∞
long	-∞	-∞	-∞	-∞	-∞	0	1	2	-∞
float	-∞	-∞	-∞	-∞	-∞	-∞	0	1	-∞
double	-∞	-∞	-∞	-∞	-∞	-∞	-∞	0	-∞
class	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞	?

The distance from type A to type B is indicated by the item (A, B) in the above table, where A is the row index and B the column index. For example, the *distance* from *byte* to *double* is the item at the row *byte* and the column *double,* i.e., 5.

The distance between class types in the above table is calculated as follows:

a) The distance between two identical class types is 0.

b) The distance between *Object* and any other class type is -∞.

c) If the class type is a *subtype* of the other type, then the distance is equal to the number of levels of *subtypes* between them. The direct *supertype* is at distance 1.

d) If none of the above is the case, but the type has *superinterfaces*, then the first *superinterface* that has a distance greater than or equal to 0 with the other type determines the distance between the two class types.

e) The distance between any two unrelated classes is -∞.

The distance between two method types, say S and T, is defined as the sum of the distances between the individual arguments of the two method types if they have the same number of arguments. Otherwise, the distance is -∞. If any single distance between an argument of the method type S and its corresponding of the method type T is -∞, then the resulting distance is -∞.

At this point, espresso has laid out the foundation for type checking.

4.3.5.3 Resolve overloaded methods

Java supports method overloading. Which version of method should be used is determined by type checking. Java Language Specification (JLS) requires the most specific method to be used. The method type distance is the measurement of the specificness of a candidate method definition for a method invocation. Whenever we type check a method invocation node, we will calculate the type distance between the method call and each possible overloaded method definition using the concept of method type distance. The method that has the smallest distance to the type of the invoked method is the most specific one, which should be used according to JLS.

In Espresso, class *ClassType* has a method *findMethod*() to look up the symbol table and to determine the most specific method declaration for a *MethodExpNode*. Type checking of *MethodExpNode* first sets up an instance of class *MethodDesc*, which contains a method type describing the signature of the method invocation expression, the name of the method, and some other fields. Then it passes this object to the initial class type on which the method is invoked to call the *findMethod* to get the most specific method definition for the method call.

Within *findMethod*, methods obtained from the symbol table with a matching number of arguments are sorted out, and among those, the most specific one is chosen by applying the *distanceTo* method of the initial method type to the type of each candidate method. Once the most specific candidate method definition is determined within a class, the search continues recursively up to the superclass. The search either stops when an ambiguity among candidate method declarations is encountered (a type check error), or when Java's primordial type Object is reached. If searching was successful, the most specific method is returned to the initial caller.

During the search up the type hierarchy, *findMethod* also takes care of requesting JIM to load required class files in order to make the appropriate type information available in the symbol table.

4.3.5.4 Type checking for primitive operators on primitive types

For all the unary and binary operators on primitive types, an initial environment is added to the symbol table. This environment defines, for each operator, its possible operand types, and the resulting primitive type. This is done by adding the primitive operator to the symbol table as an ordinary symbol, along with a method type for all possible types supported by the operator describing the types of the arguments and the result type.

To type check an expression involving a primitive operator, we only need to determine the types of its arguments, and create an appropriate method type. Then we can use the definitions given in the initial environment together with the *distanceTo* function presented above, to determine whether the expression is correctly typed and what the resulting type is.

4.3.6 Code generation

After Espresso has successfully completed the type checking it will move on to the last phase of a compiler: code generation.

Espresso uses the package *ClassGen*, which is available from BCEL. This package provides a complete infrastructure to build up a constant pool, from which at the end of the compilation, the content of a class file can be retrieved. It also includes a complete set of classes representing the instruction set of the JVM.

Generating code for particular type declaration starts by creating an instance of *ClassGen*, the top-level class of the package *ClassGen*, which is used to manage the construction of the Java class file. It provides methods to add the components of the class body and to access the class files constant pool directly if needed.

Translation proceeds by adding appropriate code for the entities of the class body. Generating code for interface declarations mainly consists of adding entries for field declarations and method declarations to the constant pool. Generating code for class declarations requires translating the Java code contained in the method body. After completing translation of all entities of the class body, the content of the generated Java class is retrieved from ClassGen, obtaining an instance of class JavaClass (contained in package JavaClass), which provides a method to dump its content to a Java class file.

4.4 Summary

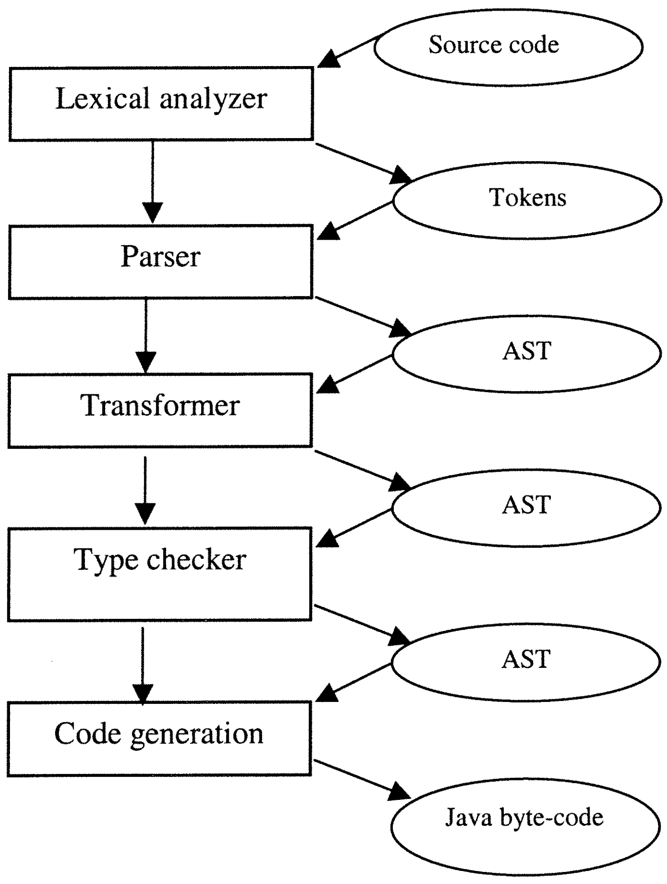
We have introduced briefly packages JavaCC, BCEL, and Espresso. We would like to summarize their functionality and the relationships among them. First, Espresso uses JavaCC to parse input Java source code against Java language grammar. If parsing successfully, JavaCC will build an internal representation of the source code, an AST. Then Espresso works on AST and uses ClassGen API to build a JavaClass object out of the AST with the help of symbol table. Then *JavaClass* API will dump Java byte code out of the *JavaClass* object and finish the compiling process.

Chapter 5

Implementation of the J compiler

The construction of the compiler for J is based on Espresso introduced in chapter 4. We will follow the natural phases of a compiler to explain the compiler construction for J. In addition to the conventional compiler construction stages, we add a transformer stage after parser, which will transform all the J extensions back to the standard Java program structure. Most of our J extensions are implemented at phases of parsing, transformation, and type checking.

The high-level flow diagram of the implementation of J compiler is depicted as below:



5.1 Lexical analyzer and syntactic parser

As we introduced in chapter 4, we use the compiler tool JavaCC to implement the front-end of our compiler for J. JavaCC combines the lexical analyzer and syntactic parser together.

First we must modify the JavaCC grammar for Java to implement all the J specific syntax. It includes modifying both the JavaCC grammar productions and the semantic actions in each relevant production.

5.1.1 J class declaration

We must change the grammar production ClassDeclaration() as:

```
ClassDeclaration() :
   ("abstract"|"final"|"public")* "class" <IDENTIFIER>
   [ "extends"  ResolvedName() ]
   [ "implements"  ResolvedNameList() ]
   "begin"
      ( ClassBodyDeclaration() )*
   "end"
```

In order to implement the new syntax for the *main* method and the factoring of static members of a class, we add two choices into the grammar production *ClassBodyDeclaration*(). For the static declaration, the match will not return any syntax tree node, but only make a mark so that the following class body declarations will know that it should add a static modifier. As to the possible *main* method in a class declaration, we must add a *mainDeclaration* production because its syntax is different from that of a normal method declaration. Within production *mainDeclaration*, we must patch the J-style main method declaration to a normal one: add modifiers *public*, *static,* and *void*, recover the type signature for the formal parameter, and return a standard method declaration node.

```
ClassBodyDeclaration() :
(
    LOOKAHEAD("static" "{")   StaticInitializer()
  |
    LOOKAHEAD(["public"|"protected"|"private"] Name() "(" )
    ConstructorDeclaration()
  |
    LOOKAHEAD("static" ":")   "static" ":"
  |
    LOOKAHEAD( MethodDeclarationLookahead() )
    MethodDeclaration()
  |
    FieldDeclaration()
  |
      MainDeclaration()
    )
```

5.1.2 If and while statements

For these statements, we remove all the syntactically unnecessary parentheses enclosing the branch condition *expression* and introduce keyword "then" for the *if* statement and use keyword "do" with *while* to make program more human readable.

```
IfStatement() :
  "if"  Expression()  "then"
      Statement(true)
  [  "else"  Statement(true) ]

WhileStatement() :
  "while"  Expression() "do"   Statement(true)
```

5.1.3 Use of traditional operators

In order to use Simula operator ":=" as assignment operator, we must change several grammar productions where the assignment operator could be used: the variable declaration (where there could be variable initialization), *AssignmentOperator*, and equality expression.

```
VariableDeclarator() :
  VariableDeclaratorId()   [ ":=" VariableInitializer() ]

Assignment() :
 PrimaryExp()  AssignmentOperator()  Expression() )

AssignmentOperator() :
  ( ":=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<="
  | ">>=" | ">>>=" | "|=" | "&=" | "^=" )

EqualityExp()  :
  InstanceOfExp()  [ ":="  InstanceOfExp() ]
```

As to the implementation of content identical operators "=" and "!=", it is relatively simpler, we must add two operators "=" and "!=" into the IdentityExp production.

```
IdentityExp() :
  EqualityExp() ( ("=="|"!=="|"="|"!=") EqualityExp() )*
```

Though, the syntactic modification is trivial, it needs further processing in the stage of type checking.

5.2 Transformer visitor

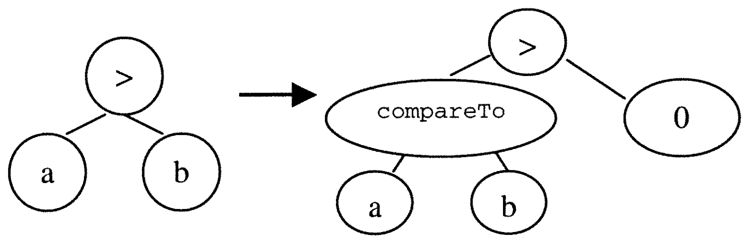
After parsing a J program, we get an abstract syntax tree and a symbol table organized as a hash table. They retain all the information of the original code. Before type checking, we use a transformer visitor to traverse the resulting syntax tree and transform the J extensions back to standard Java program structure.

5.2.1 Object order comparison

This feature is handled when transformer visitor reaches a *RelationalExpNode* such as $a > b$. First, it will check the type of left and right expressions of this node. If both of them are of class type and not of primitive type, then it knows that it must do transformation on this node.

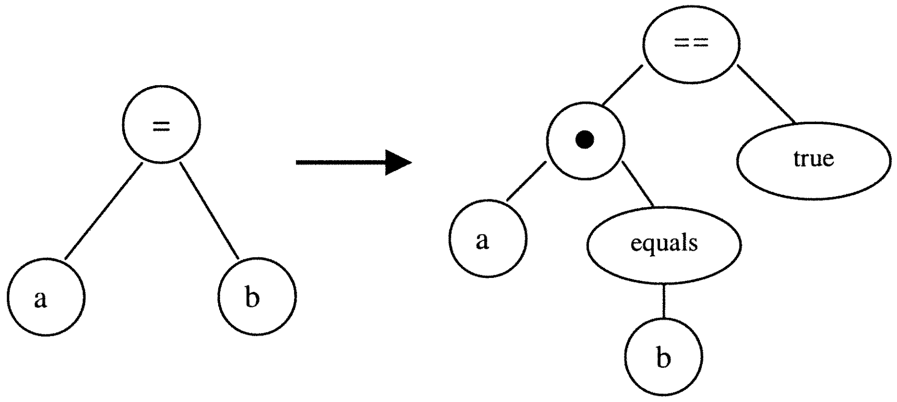
Before any action is taken, the transformer checks whether the left and right expressions of this node are type compatible. If not, visitor will throw a type checking error exception. Otherwise, it will transform this node into a *MethodExpNode*. In order to make this feature available, the involved class must have implemented the method *compareTo*() of interface Comparable.

After all these checkings, the transformer starts to do the transformation on syntax tree. For example, the action for expression `a > b` is to replace it with expression *a.compareTo(b)* > *0* as shown in the diagram that follows.



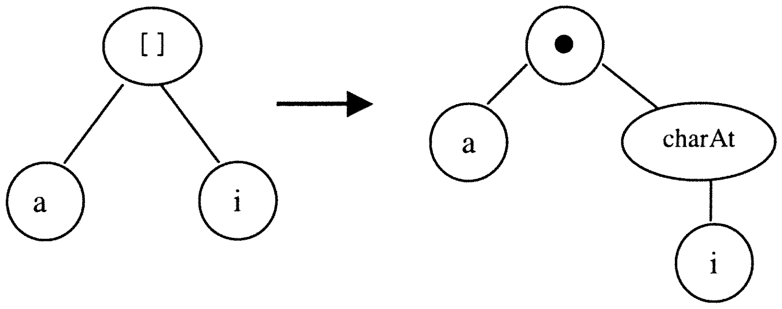
5.2.2 Content identical comparison

We use two operators ("=" and "!=") and a new expression node class *IdentityExpNode* for this purpose. When constructing the *IdentityExpNode*, for example, `a = b`, we mark the expression node to distinguish different operators. When visiting this node, transformer will check the type of both operands and the operator type first. If *a* and *b* are not of class type, then a type checking error will be thrown. Otherwise, if the operator is "=" or "!=", then the node will be transformed into another *identityExpNode* whose left operand is a *MethodExpNode* `a.equals(b)` and the right is a *BooleanLiteral ExpressionNode* of value true, i.e., the content identical comparison of class object will become: `a.equals(b) == true` or `a.equals(b) != true` depending on the type of operator. Note that the class involved must have the method *equals*() implemented according to your specific requirements. Otherwise the *equals*() method of Object will be used. The transformation is demonstrated as follows:



5.2.3 Indexing for String and Vector

When visiting the *ArrayExpNode*, for example *a[i]*, type checker will check the type of the left operand of this node. If it is of type *String*, then this node will be replaced by *MethodExpNode*: *a.charAt(i)*. If it is of type *Vector*, then it will be transformed into a *MethodExpNode*: *a.elementAt(i)*. The transformation for *[]* on *String* is demonstrated as follows:



Note that in order to do this transformation, the visitor must go up a level on the syntax tree because *MethodExpNode* and *ArrayExpNode* are not type compatible. The way we process the transformation is that we add a member variable, `resultExp_d`, in class *ExpressionNode*, which is the super class of all expression nodes. When we detect that the current *ArrayExpNode* needs transformation, we create a methodExpNode and assign it to variable *resultExp_d*. When the visitor sees this node for the next time, it will check variable *resultExp_d* first. If it is not *null*, then the current node will be replaced by `resultExp_d`.

5.3 Type checking

Now we must check for the semantic validity of the program, which is the main purpose of type checking. Furthermore, some of our J extensions to Java are implemented at this phase.

In addition to the normal type checking, the type checker must detect whether a program structure is in standard Java or in J extension. If it is the latter, we need to annotate the AST with some additional information.

5.3.1 Automatic type casting

This feature is mainly handled by the class *AssignmentNode*. We assume that the users are conscious of what they are doing and therefore use a very simple algorithm. When type checker visits this node, it will first find out the type of both left and right expressions of the assignment. If the types of both sides are different and the distance is greater than 0, it means that they can be inter-assignable. Then we cast the right side to the type of the left side by default. This rule applies to both primitive types and class types.

In case there is a cast needed, we will create a *CastExpNode* with the type of the left side and the expression of the right side, and the newly created expression will replace the original right side expression.

5.3.2 The Method selection

In order to relax the method select rules in J, we must redefine the type distance matrix in chapter 4. The main modification is that we merge integral series of primitive and floating type series into one. The new type distance matrix is shown in the following table:

Table 5.1 Type distance matrix of J

	boolean	byte	short	char	int	long	float	double	class
boolean	0	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞
byte	-∞	0	1	2	3	4	5	6	-∞
short	-∞	1	0	1	2	3	4	5	-∞
char	-∞	2	1	0	1	2	3	4	-∞
int	-∞	3	2	1	0	1	2	3	-∞
long	-∞	4	3	2	1	0	1	2	-∞
float	-∞	5	4	3	2	1	0	1	-∞
double	-∞	6	5	4	3	2	1	0	-∞
class	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞	?

All the types are represented by a class each, for example, class *IntType* for int, *FloatType* for float, etc. This matrix is implemented by the method *distanceTo*() in each type class. Following is the method in class *DoubleType*:

```
public int distanceTo( Type other ) {
    int result = Integer.MIN_VALUE;
    if ( other instanceof ByteType ) {
      result = 6;
    }
    else if ( other instanceof CharType ) {
      result = 5;
    }
    else if ( other instanceof ShortType ) {
      result = 4;
    }
    else if ( other instanceof IntType ) {
      result = 3;
    }
    else if ( other instanceof LongType ) {
      result = 2;
    }
    else if ( other instanceof FloatType ) {
      result = 1;
     }
    else if ( other instanceof DoubleType ) {
      result = 0;
    }
    return result;
}
```

While for the standard Java, the counterpart is:

```
public int distanceTo(Type other) {
     int result = Integer.MIN_VALUE;
     if ( other instanceof FloatType ) {
       result = 1;
      }
     else if ( other instanceof DoubleType ) {
       result = 0;
     }
     return result;
}
```

We can see from the above example that in Java, double is related to double and float only, but in J, double is related to all the primitive types (except for *boolean*).

When looking up for a method definition, J compiler will see all the primitive types compatible with the new type distance matrix. First it will check the number of arguments of a method call and the number of formal parameters of a method declaration. If they are not the same, then J compiler will reject the method declaration. Otherwise, it could accept it as candidate if it can pass some further type checking. Therefore, it will accept the following sample program given in chapter 2:

```
class Test
begin
      int sum;
      public void add( short item ) {
            sum += item;
      }
static:
main(args){
      Test t := new Test();
      t.add(73);
}
end
```

5.4 Code generation

Traditionally, this is the most complex part of compiler. However, when we compile the J program, we retain the internal structure (i.e., AST) of Java. The transformer pass in our compiling process could be considered part of code generation. It transforms a J-specific AST to a standard Java AST so that we can reuse the Espresso's code generation facilities for Java to generate Java byte-code for J.

For the purpose of compatibility and ease of comparison, we implement a pretty printer that can produce equivalent Java code for each J program.

Chapter 6

Testing and evaluating of J

In this chapter, we test and evaluate J. To test the system, we choose a set of programs that exemplify all the new features we implemented. The examples are compiled first with J then run with the command *java*. To evaluate J, we compare the J version of a program with the equivalent Java code and count the number of tokens in code of both versions as a measure of complexity.

6.1 Tests

To check out J we choose 5 programs that cover all our language modifications. Following are what will be tested:

program 1: Syntax modifications

program 2: Object order and equality comparison

program 3: Indexing

program 4: Type cast

Program 5: Method selection.

6.1.1 Syntax modifications

We show a small J code example that uses almost all the J specific syntax like the new assignment operator, static keyword factoring, the main method declaration, class declaration, etc.

```
import java.io.*;

public class Test1
begin
    int i := 0;
static :
    int j := 1;
    String s1 := "bc";
    public void print(String msg){
        System.out.println(msg);
    }

main (args){
    String s2 := "abcdef".substring(1, 2);
    while j <= 2 do {
        if one = two then
            print("s1 equals to s2");
        else if s1 == s2 then
            print("s1 and s2 are the same");
        j++;
    }
    Integer v1 := new Integer(123);
    Integer v2 := new Integer(123);
    if v1 = v2 then
      print("Integers equal");
    if v1 == v2 then
      print("Integers identical");

}
end
```

6.1.2 Test of object order and equality comparison

Class Point is used to test the implementation of object order and equality comparison. It has two attributes: *x* and *y*, for the coordinates. We define a method *distance* that measures how far a point is from the origin and is used to specify the *compareTo* method so that we can overload comparison operators <, >, >=, and <= upon Point objects.

```
import java.io.*;

class Point
begin
    int x, y;
    Point(int x, int y){
        this.x := x;
        this.y := y;
    }
    int distance(){ // from origin
        return x * x + y * y;
    }
    int compareTo(Point other){
        if this.distance() > other.distance() then
            return 1;
        else if this.distance() < other.distance() then
            return -1;
        else
            return 0;
    }
end

public class Test2
begin
static :
    public void print(String msg){
        System.out.println(msg);
    }
main (args){
    Point p1 := new Point(10, 5);
    Point p2 := new Point(5, 10);
    if p1 > p2 then
```

```
            print("P1 is further to the origin than p2")

        if new Integer(12) > new Integer(5) then
            print("12 is greater than 5");
        if "abc" > "zzz" then
            print("String abc is greater than string zzz");
}
end
```

6.1.3 Test of String and Vector indexing

Test3 is a simple J class that has a static *print* method and a *main* method. The *main* method defines a *String* variable and a *Vector* variable and then enumerates their contents with indexing of String and Vector.

```
import java.io.*;
import java.util.*;

public class Test3
begin
static :
    public void print(String msg){
        System.out.println(msg);
    }
main (args){
    String str := new String("Hello ");
    Vector v := new Vector();
    v.addElement(str);
    v.addElement(new String("world"));
    for (int i := 0; i < str.length(); i++)
        print(str[i]);

    for (int i := 0; i < v.size(); i++)
        str := v[i];
}
end
```

6.1.4 Test of automatic type casting

Test4 includes a variable definition and a statement that are illegal in Java but legal in J. In the definition *float f:= 0.9, f* is a float but 0.9 a double. Therefore they are not type compatible in Java. But in J, when compiling this definition, 0.9 will be cast into a float implicitly so that J will accept this variable definition. A similar procedure is applied to the assignment expression *s := s + 1* .

```
import java.io.*;

public class Test4
begin
static :
    public void print(String msg){
        System.out.println(msg);
    }
main (args){
    short s ;
    float f := 0.7;    //0.7 is a double
    s := s + 1;        //s: short; s+1: int
    print(" You should see no error message!");
    print(" f = " + f + "  s = " + s);
}
end
```

6.1.5 Test of new method selection algorithm

The class Test5 overloads the method *count* several times and then the method is called with parameters that do not match exactly the formal parameters of any method definition. It will test whether our new method selection algorithm works or not. The operation details are explained by the comments in the program.

```
import java.io.*;

public class Test5
begin
static :
    public void count(int i, int j){
        print("(int, int)");
    }
    public void count(int i, long j){
        print("(int, long)");
    }
    public void count(double i, double j){
        print("(double, double)");
    }
    public void print(String msg){
        System.out.println(msg);
    }

main (args){
    short s := 1;
    int i := 2;
    long l := 23L;
    float f := 0.9f;
    double d := 0.9;

    /* casting: (short->int, short->int) */
    count(s, s);

    /* (int, short -> int) */
    count(i, s);

    /* (int, long) */
    count(i, l);

    /* (long -> int, long) */
    count(l, l);

    /* (float -> double, float -> double) */
    count(f, f);

    /* (float-> double, double) */
```

```
        count(f, d);

        /* (long -> int, float-> long) */
        count(l, f);

    }
    end
```

In the above program, we use the comments to indicate the expected behavior of J. When selecting the method, we use only one criterion: the *specificness* of a method (defined in chapter 4) based on the type distance matrix given in table 5.1.

When we run the above test programs with our J compiler, it does accept these programs and behaves successfully as we expected.

6.2 Comparisons with Java

An important purpose of J is to simplify the code writing. We will compare J programs to their Java equivalents to see if J achieves our design goal.

The comparison is done in two ways: first we take a J program and use pretty printer of J to output its Java equivalent. Then we compare the J program with the generated Java program (though we may manually make some minor modifications to the output Java code, for example, changing *java.lang.String* to *String).* Second we pick up some Java code examples from textbooks on data structures because there are many good examples of complex algorithms. Then we rewrite them in J and compare them with the original Java code.

The criteria of the comparison are the tokens count and the characters count. Generally, they are good indicators of program simplicity. We will do some simple statistics on tokens and characters to see which one is simpler.

6.2.1 From J to Java

We pick up all the examples in section 6.1 and rewrite them in Java. The Java counterparts are listed in appendix A. We count the number of tokens and characters for each of them and then calculate the benefit of J by counting how many tokens and characters are saved. The comparison results are listed in tables 6.1 and 6.2.

Table 6.1 Token number statistics

Examples	Tokens count of Java	Tokens count of J	Number of J features	Benefit of J	Improvement %
Test 1	287	270	8	17	5.9
Test 2	175	165	5	10	5.7
Test 3	106	97	4	9	8.5
Test 4	96	85	4	11	11.5
Test 5	165	135	4	30	18.2
Average					9.9

Table 6.2 Character number statistics

Example	Characters count of Java	Characters count of J	Benefit of J	Improvement %
Test 1	417	398	19	4.6
Test 2	584	557	27	4.6
Test 3	343	314	29	8.5
Test 4	220	196	24	10.9
Test 5	476	399	77	16.1
Average				8.9

From the above statistics, we can see that J has 9.9% fewer tokens and 8.9% fewer characters in average than Java does.

The results are consistent with the design of J since J is only an extension to Java. Whenever there is a J specific feature involved, the J code always has fewer tokens than its Java counterpart. If there is no J specific feature, the J code is always the same as that of Java.

6.2.2 From Java to J

We want to know in real Java programs, how often the J specific features could appear and how much we can benefit from J. Therefore we pick up some representative

Java code examples and rewrite them. For objectiveness and randomness, we pick up some textbooks on data structures using Java, such as [Weiss 99], [Sta 99], and [Gt 01]. We use [Weiss 99] only because it implemented all the sorting algorithms discussed in it and is therefore suitable for our test. We choose the Java programs for insertion sort, shell sort, heap sort, merge sort, and quick sort, and rewrite them in J. The statistics result is shown in tables 6.3 and 6.4:

Table 6.3 Token statistics for all sort examples

Example	Tokens count of Java	Tokens count of J	Number of J features	Benefit of J	%
Insertion sort	87	81	2	6	6.9
Shell sort	106	100	2	6	5.7
Heap sort	311	298	7	15	4.8
Merge sort	162	158	6	10	6.2
Quick sort	303	275	18	38	12.5
Average					7.2

Table 6.4 Character statistics for all sort examples

Example	Characters count of Java	Characters count of J	Benefit of J	%
Insertion sort	161	144	17	10.6
Shell sort	204	191	13	6.4
Heap sort	457	430	27	6.0
Merge sort	859	845	14	1.6
Quick sort	1596	1459	137	8.6
Average				6.7

This test result also shows that all the examples contain more than one occurrence of J specific features and we can often have a gain in terms of tokens count and characters count: J has 7.2% fewer tokens and 6.7% fewer characters than Java does.

From the comparisons in the above two subsections, we can conclude that the overall average of the gain of J is: 8.8% fewer tokens and 7.8% fewer characters.

To show the improvement of readability, we rewrite in J the quick sort program in

[Weiss99] and list both the Java version and J version in appendix B. In the following table we list only a code segment that J can simplify most strikingly:

J code	
	```
	. . .
	if a[center] < a[left] then
	  swapReferences(a, left,center);
	if a[right] < a[left]  then
	  swapReferences(a, left, right);
	if a[right] < a[center]  then
	  swapReferences(a, center, right);
	. . .
	```

Java code	
	```
	. . .
	if(a[center].compareTo(a[left]) < 0)
	  swapReferences(a, left, center);
	if(a[right].compareTo(a[left]) < 0)
	  swapReferences(a, left, right);
	if(a[right].compareTo(a[center])<0)
	    swapReferences(a, center, right);
	. . .
	```

Note that the comparison in J between two *Comparables* is just like comparing two numbers. The advantage of J is double-folded: it can both save lots of typing and make the algorithm much easier to understand.

6.3 Some comments on J

There are some limitations on J that should be noted. First, some features like the indexing of *String* and *Vector* is only partially implemented. The indexing can only appear at the right hand side of an assignment expression. It may cause some non-uniformity and confusion.

Second is the precision loss problem. We allow type conversion between any two types (say A and B) if there is a path from A to B in the type chains, for example, we allow type conversion from *long* to *byte*. Some bits of the *long* will be lost and the action may behave different from our expectation. When occurred in the new method selection algorithm, the situation could be improved (not completely solved) by redefining a new type distance matrix as follow:

	boolean	byte	short	char	int	long	float	double	class
boolean	0	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞
byte	-∞	0	1	2	3	4	5	6	-∞
short	-∞	6	0	1	2	3	4	5	-∞
char	-∞	6	5	0	1	2	3	4	-∞
int	-∞	6	**5**	4	**0**	1	2	**3**	-∞
long	-∞	6	5	4	3	0	1	2	-∞
float	-∞	6	5	4	3	2	0	1	-∞
double	-∞	6	5	4	3	2	1	0	-∞
class	-∞	-∞	-∞	-∞	-∞	-∞	-∞	-∞	?

In the above matrix, whenever the casting is going to lose precision, the distance is greater than that of any compatible casting. For example, the distance from *int* to *short* (5) is even greater than that from *int* to *double* (3). This is the matrix used in our J compiler.

For the automatic type casting, we do not automate the type conversion for complicated expressions because it maybe very difficult to derive (not just guess) the correct target type from the expressions themselves. Programmers have to do it explicitly when coding the program.

At the beginning of our research, we aimed to make the primitive types and their corresponding wrapper classes interchangeable. After a second thought, we gave up this idea. The first reason is that it will entail lots of details while introducing no new concept. For example, suppose we want to do operation like: `new Integer(10) + 2`, we must check all the possibilities when type checking the addition expression. If both two operands are of *Integer*, we must transform them into two *int* numbers; if only one of Integer, we transform it into an *int*; and if one operand is of type Integer and the other of type *long*, then we must change the *Integer* into a *int* number first and then cast it to a *long* number, etc. The same procedure applies as well to the other primitive operators. But the intent here is nothing more than transforming an *Integer* object into an *int* number. Second, mixing an object with a primitive number contrasts bluntly to the object-oriented paradigm and may introduce concept confusion. Therefore, we opt not to do anything on the wrapper classes.

6.4 Conclusion

From the above tests and comparisons, we can conclude that we have successfully implemented the language J as designed, and J simplifies the code writing and improves the code readability. How much it can save depends on how many J- specific features occurs in the program. The more occurrences, the more J can simplify the code writing. On average, J has 8.8% fewer tokens and 7.8% fewer characters than Java does.

Chapter 7

Conclusions

In this thesis, we designed a new language called J to simplify the Java by reducing the typing work and making the program more readable. We began with an overview of Java and its evolution through version 1.0 to version 1.3. Then we discussed Java's new language concepts that explain why Java achieved success so rapidly. Briefly, Java is portable so that an application can be developed once and run everywhere; Java has such a huge library that users can reuse components and develop applications more easily with it than with most other languages; and finally, by means of garbage collection, no explicit pointer, and exception handling, Java can ensure the reliability of applications.

On the other hand, Java has its own weaknesses. Its syntax is both verbose and cryptic. Java can confuse users by using the same operator "==" to test both object identity and primitive value equality; it requires too much type casting; and its method selection algorithm is too rigid when handling overloaded methods with primitive parameter types.

The language that we designed and implemented, J, simplifies the external form of Java but retains its internals. J simplifies the Java syntax by introducing keyword factoring (*static*), a simplified main method declaration, and changing the syntax for *if* and *while* statements. On the semantic front, we overload order comparison operators and the indexing operator ([ ]) for String and Vector. These modifications simplify the writing and the reading of programs. J automates some type casting to minimize the explicit use of type casting operations in Java. The most complex improvement that J introduces is the relaxing of the method selection algorithm of Java, especially when the parameter types are primitive. For this purpose, J redefines the type distance matrix between primitive types and inter-relates the floating-point types and integral types.

Our implementation uses existing tools and packages found on the Web in particular: JavaCC, BCEL and Espresso. We gave a short introduction to their functionality and examples of their use so that readers can follow this thesis smoothly.

The implementation of the J compiler is also based on the JavaCC grammar for Java. Though we modified the JavaCC grammar file slightly and the associated embedded semantic actions. We reused the entire class hierarchy for the construction of an abstract syntax tree at the phase of parsing. An important addition was a J-specific compiler phase using a "visitor", which transforms all the J-specific syntax features on the syntax tree into standard Java. It was followed by a phase of type checking, where the automatic type casting and new method selection algorithm were implemented. At the last phase, we simply reused the code generator from Espresso because when we reach this stage every program constituent is in standard Java.

Tests done in chapter 6 demonstrated that a program written in J was both shorter and clearer than one in Java. On average, J contains 8.8% fewer tokens and 7.8% fewer characters than Java does. It does not only have fewer tokens but also reduces potential coding errors by making J code more human-readable.

References

[Ac96]	M.Abadi, and L. Cardelli, *A Theory of Objects*, Springer-Verlag, 1996.
[Ag98]	K. Arnold, and J. Gosling, *The Java programming Language*, Addison-Wesley, 2nd Ed., 1998.
[Asu 86]	A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
[Bcel]	Markus Dahm, *Welcome to the Byte Code Engineering Library 4.4.0*, http://bcel.sourceforge.net/, 2001.
[Bg 96]	T.J. Bergin, and R. G. Gibson, *History of programming Languages*, Addison-Wesley, 1996.
[Bm62]	R A. Brooker, and D. Morris, *The compiler-compiler*, in *Annual Review in Automatic Programming* Vol. **3** (1962), Pergammon, London.
[Cox 86]	B. Cox, *Object-oriented Programming: an evolutionary approach*, Addison-Wesley, 1986.
[Espresso]	K. Doerig, *A Java compiler written in Java*, Boston University, http://types.bu.edu/Espresso/report/Espresso.html
[Fwh 92]	D. P. Friedman, M. Wand, and C. T. Haynes, *Essentials of Programming Languages*, MIT Press, 1992.
[Gj 87]	C. Ghezzi and M. Jazayeri, *Programming language Concepts*, 2nd Ed., Wiley, 1987.

[Gjs 96] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[Gt 01] Michael T. Goodrich, and Roberto Tamassia, *Data structures and algorithms in Java*, 2nd ed.. John Wiley, 2001.

[Hoare 73] C. A. R. Hoare, *Hints on Programming Language design*, in *Proceedings of ACM Symposium on Principles of Programming Language*, CAN Press, 1973.

[Java 10] Sun Microsystems Inc., *Archive: Java Development Kit (JDKTM), v 1.0.2.* http://java.sun.com/products/archive/jdk/1.0.2/index.html

[Java 11] Sun Microsystems Inc., *Java Platform 1.1 API Specification.* http://java.sun.com/products/jdk/1.1/docs/api/packages.html

[Java 12] Sun Microsystems Inc., *Java 2 Platform, Standard Edition, v1.2.2 API Specification.* http://java.sun.com/products/jdk/1.2/docs/api/overview-summary.html

[Java 13] Sun Microsystems Inc., *Java 2 Platform, Standard Edition, v 1.3.1 API Specification.* http://java.sun.com/j2se/1.3/docs/api/overview-summary.html

[Javacc] Metamata, Inc., *Java Compiler Compiler (JavaCC) - The Java Parser Generator.* http://www.webgain.com/products/java_cc/

[Javax] J. Rosenberg, *An approachable examination of Java, Javabeans, Javascript and all the related Java technologies.* http://developer.netscape.com/docs/wpapers/javax/javax.html

[Kr 88] B. W. Kernighan, and D. M. Ritchie, *The C Programming Language*, 2nd Ed., Prentice-Hall, 1988.

[Lmb 95] J. R. Levine, T. Mason, and D. Brown, *Lex & Yacc*, 2nd edition, O'Reilly & Associates, 1995.

[Ly 97] T. Lindholm, and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.

[Mak 91] R. Mak, *Writing compilers and interpreters: an applied approach*, John Wiley & Sons, 1991.

[Muc 97] S. S. Muchnick, *Advanced compiler design implementation*, Morgan Kaufmann Publishers, 1997.

[Sta 99] T. A. Standish, *Data structure in Java*, Addison-Wesley, 1999

[Strou 94] B. Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994

[Tenn 81] R. D. Tennent, *Principles of Programming Languages*, Prentice-Hall, 1981.

[Thim 99] H. Thimbleby, *A critique of Java.* Software-Practice and Experience, vol. **29**, no.5, pp. 457-78, 1999.

[Watt 90] D. A. Watt, *Programming Language Concepts and Paradigms*, Prentice-Hall, 1990.

[Weiss 99] M. A. Weiss, *Data structures & Algorithm analysis in Java.* Addison-Wesley, 1999.

Appendix A

Equivalent Java programs for comparison

Test1.java

```
import java.io.*;

public class Test1
{
    int i = 0;
    static int j = 1;
    static  String s1 = "bc";

    public static void print(String msg){
        System.out.println(msg);
    }
    public static void main (args){
      String s2 = "abcdef".substring(1, 2);
      while (j <= 2) {
          if (one = two)
            print("s1 equals to s2");
          else if (s1 == s2)
            print("s1 and s2 are the same");
          j++;
      }
      Integer v1 = new Integer(123);
      Integer v2 = new Integer(123);
      if( v1.equals( v2))
          print("Integer equals");
      if ( v1 == v2)
          print("Integer identical");

    }
}
```

Test2.java

```
import java.io.*;

class Point {
    int x, y;
    Point(int x, int y){
        this.x = x;
        this.y = y;
    }
    int distance(){
        return x * x + y * y;
    }
    int compareTo(Point other){
        if (this.distance() > other.distance())
```

```
            return 1;
        else if ( this.distance() < other.distance())
            return -1;
        else
            return 0;
    }
}

public class Test2 {
    public static void print(String msg){
        System.out.println(msg);
    }
    public static void  main (args){
      Point p1 = new Point(10, 5);
      Point p2 = new Point(5, 10);
      if (p1.compareTo (p2) > 0 )
          print("P1 is further to the origin than p2");

      if( (new Integer(12)).compareTo(new Integer(5) ) > 0 )
          print("12 is greater than 5");
      if ("abc".compareTo("zzz") > 0)
          print("String abc is greater than string zzz");
    }
}
```

Test3.java

```
import java.io.*;
import java.util.*;

public class Test3
{
    public static void print(String msg){
        System.out.println(msg);
    }
    public static void main (args){
      String str = new String("Hello ");
      Vector v = new Vector();
      v.addElement(str);
      v.addElement(new String("world"));
      for (int i = 0; i < str.length(); i++)
          print(str.charAt(i));

      for (int i = 0; i < v.size(); i++)
          str = (String) v.elementAt(i);
    }
}
```

Test4.java

```
import java.io.*;

public class Test4
{
    public static void print(String msg){
        System.out.println(msg);
    }
    public static void main (args){
      short s ;
      float f = (float) 0.7;
      s := (short) (s + 1);
      print(" You should see no error message!");
      print(" f = " + f + "  s = " + s);
    }
}
```

Test4.java

```
import java.io.*;

public class Test5
{
    public static void count(int i, int j){
        print("(int, int)");
    }
    public static  void count(int i, long j){
        print("(int, long)");
    }
    public static void count(double i, double j){
        print("(double, double)");
    }
    public static void print(String msg){
        System.out.println(msg);
    }

    public static void main (args){
      short s = 1;
      int i = 2;
      long l = 23L;
      float f = 0.9f;
      double d = 0.9;

      count((int)s, (int)s);

      count(i, (int)s);

      count(i, l);

      count((int)l, l);

      count((double)f, (double)f);

      count((double)f, d);

      count((int)l, (long)f);
    }
}
```

Appendix B

Quicksort programs

1. Java code for QuickSort

```
public final class QuickSort {
    private static final int CUTOFF = 3;

    public static void quicksort(Comparable [ ] a){
        quicksort( a, 0, a.length - 1 );
    }

    public static final void swapReferences(Object[] a, int index1, int
index2){
        Object tmp = a[index1];
        a[index1] = a[index2];
        a[index2] = tmp;
    }

    private static Comparable median3(Comparable[] a, int left, int
right){
        int center = (left + right) / 2;
        if(a[center].compareTo(a[left]) < 0)
            swapReferences(a, left, center);
        if(a[right].compareTo(a[left]) < 0)
            swapReferences(a, left, right);
        if(a[right].compareTo(a[cente ]) < 0)
            swapReferences(a, center, right);

        // Place pivot at position right - 1
        swapReferences(a, center, right - 1);
        return a[right - 1];
    }

    private static void quicksort(Comparable [] a, int left, int right){
         if(left + CUTOFF <= right) {
             Comparable pivot = median3(a, left, right);

             // Begin partitioning
             int i = left, j = right - 1;
             for( ; ; ) {
                while(a[ ++i ].compareTo(pivot) < 0) { }
                while(a[--j].compareTo(pivot) > 0) { }
                if(i < j)
                   swapReferences(a, i, j);
                  else
                   break;
              }
             swapReferences(a, i, right - 1);   // Restore pivot
             quicksort(a, left, i - 1);    // Sort small elements
             quicksort(a, i + 1, right);   // Sort large elements
          }
          else  // Do an insertion sort on the subarray
             insertionSort(a, left, right);
    }

    private static void insertionSort(Comparable [] a, int left, int
right){
          for(int p = left + 1; p <= right; p++)
          {
              Comparable tmp = a[p];
              int j;
              for(j = p; j > left && tmp.compareTo(a[j-1]) < 0; j--)
                    a[j] = a[j - 1];
                a[j] = tmp;
            }
       }
  }
```

2. Quicksort in J

```
public final class QuickSort
begin
static:
      public void quicksort(Comparable[ ] a){
            quicksort(a, 0, a.length - 1);
       }

      private final int CUTOFF := 3;

      public final void swapReferences(Object [] a, int index1, int
index2){
            Object tmp := a[index1];
            a[index1] := a[index2];
            a[index2] := tmp;
       }

      private Comparable median3(Comparable [] a, int left, int right){
                  int center := (left + right) / 2;
                  if a[ center ] <  a[left] then
                swapReferences(a, left, center);
                  if a[right] < a[left]  then
                swapReferences(a, left, right);
                  if a[right] < a[center]  then
                swapReferences(a, center, right);

                  // Place pivot at position right - 1
                  swapReferences(a, center, right - 1);
                  return a[right - 1];
       }

       private void quicksort(Comparable [ ] a, int left, int right){
           if left + CUTOFF <= right then
           {
              Comparable pivot := median3(a, left, right);

                // Begin partitioning
              int i := left, j := right - 1;
              for( ; ; ){
                  while a[++i] < pivot do { }
                  while a[--j] > pivot do { }
                  if( i < j )
                      swapReferences(a, i, j);
                        else
                            break;
              }

             swapReferences(a, i, right - 1);   // Restore pivot
                  quicksort(a, left, i - 1);  // Sort small elements
                  quicksort(a, i + 1, right); // Sort large elements
            }
            else  // Do an insertion sort on the subarray
                insertionSort(a, left, right);
      }

      private void insertionSort(Comparable [] a, int left, int right){
           for(int p := left + 1; p <= right; p++) {
               Comparable tmp := a[p];
               int j;

               for(j := p; j > left && tmp < a[ j - 1 ]; j--)
                   a[j] := a[j - 1];
               a[j] := tmp;
           }
      }
end
```