Espresso, A Java compiler written in Java

Karl Doerig, Boston University

A report that presents Espresso, a Java compiler developed at BU during a directed study in advanced compiler design. The main goal is to introduce the basic concepts and the architecture of Espresso, in order to provide a basis for future work.

Espresso, a Java compiler written in Java is presented. It was developed in the course of a directed study in advanced compiler design over a period of about 3 months under the supervision of Prof. A. Kfoury. The main contributors to design and implementation up to the current version 0.3 are Santiago M. Pericas (email: santiago@cs.bu.edu) and Karl Doerig (email: kdoerig@cs.bu.edu)

Espresso’s main purpose is to serve as a workbench for students interested in applying their theoretical knowledge from programming languages and compiler design and construction in the context of a compiler for a modern object oriented programming language.

To enable and encourage to continue the work started, the overall architecture and important design aspects are documented.

A detailed list of language features that are currently missing or only partially available is compiled.

Suggestions for improvements for both functionality and performance are made.

A short guide on how to make Espresso available for use and development is given.
Table of Contents

1.0 Introduction 3
  1.1 Goals 4
  1.2 Results 4
  1.3 Glossary 5
  1.4 Conventions 5

2.0 Architecture 5
  2.1 Overview 6
  2.2 The main class Espresso 7
  2.3 The abstract syntax tree (AST) 8
  2.4 The symbol table 8
  2.5 JavaImportManager JIM 11

3.0 Parsing and constructing the AST 12
  3.1 Keeping track of actual package, class, method 13
  3.2 Local scopes 13
  3.3 Wrapping of control statements 14
  3.4 Adding a default constructor 14
  3.5 Non static initializers for fields 15
  3.6 Static initializers, and initializers for static fields 15
  3.7 Verifying closure of the compiled unit 15

4.0 Type checking 16
  4.1 Espresso’s type hierarchy 16
  4.2 The distance metric 19
  4.3 Type checking for primitive operators on primitive types 21
  4.4 Finding the most specific method 21
  4.5 Finding the right field 22
  4.6 Type checking of initializers for variables 22

5.0 Code generation 22
  5.1 The overall translation picture 23
  5.2 Translating a method body 23
  5.3 The role of Espresso’s type hierarchy 24
  5.4 Controlling translation for lvalues and rvalues 25
  5.5 Lazy evaluation of boolean expressions, backpatching and synthesizing 28
  5.6 Translating continue statement 31
  5.7 Translating break statement 31

6.0 Proposals for future work 31
  6.1 Missing or incomplete features 32
  6.2 Optimization 38
  6.3 Improvements to error reporting 38
  6.4 What to pick first 39

7.0 How to get involved in the project 39
  7.1 How to get access to Espresso’s most recent source code 39
  7.2 How to obtain a particular version of Espresso from CVS 40
  7.3 What are the rules for committing changes back to CVS 40
  7.4 Espresso’s directory structure 40
1.0 Introduction

Java, an object oriented programming language developed at *Sun Micro Systems* was first released in 1996. It quickly became one of the most widely used programming language in industry, in particular for the client side programming of web based internet and intranet applications.

The strength of Java lies in a carefully designed language that includes most of the benefits provided by C++, while at the same time being more compact and consistent in its design. Among others, one major reason for its success was the standardization of the Java compiler and the portability of compiled class files across platforms and operating systems.

The Java language as specified in its current version 1.1 added only one major new feature to the grammar of the language presented in 1996, namely *inner classes*.

Espresso, a Java compiler developed at Boston University in a directed study project, provides a compiler for a state of the art object oriented programming language to students who are interested to turn their theoretical knowledge from graduate level programming language and compiler design and construction courses into practical application and improvement of a modern programming language compiler. Because Espresso is an open project and contributions are highly encouraged.

In its current version, Espresso grammatically conforms to the Java language specification version 1.0.

Section 2 first presents the overall architecture of Espresso. It then introduces to design and concepts of the abstract syntax tree AST, the SymbolTable, and the JavaImportManager, a component responsible for looking up and loading compiled Java classes into Espresso’s environment.

Section 3 describes some general concepts applied during parsing and construction of the AST. It also documents the handling of special cases in this compilation step.

Section 4 introduces to type checking and highlights key concepts around which type checking is built up. It includes a detailed description of the type hierarchy and introduces to an interesting metric discovered by Santiago M. Pericas. This metric is central during type checking, because it allows to deal with the various situations involving primitive types, reference types and method types in a uniform way.

Section 5 presents the overall architecture of the code generation step and describes important concepts applied in this phase.

Section 6 contains a collection of proposals for future work. It first contains a list of Java language features that are currently only partly type checked, and/or for which code generation is missing or incomplete. It also points out possible improvements of the generated code by applying appropriate optimization techniques. Finally it makes some suggestions to improve error reporting.
Section 7 gives some basic information on Espresso’s project structure. It also defines the procedure you have to follow if you want to contribute to Espresso’s further improvement.

1.1 Goals

The main motivations and reasons for this project can be summarized as follows:

- Provide a modern and attractive platform for future work and experimentation to graduate students of both, programming languages and compiler design and construction.
- Open an opportunity to apply some of the technology from the Church Project, and the OO seminar at Boston University to a industry standard OO programming language.
- Demonstrate that a compiler for a major part of the Java language could be constructed in a one semester project involving 2 students.
- Show that the constructed compiler would compile in comparable time and would generate code of similar quality as Sun’s javac compiler does.
- Explore the possibilities of modern compiler construction tools applied to the compilation of a modern object oriented programming language.

1.2 Results

Most of the implementation oriented objectives have been reached. A first version of Espresso was available for testing and performance evaluation after about 3 month of hard work in May 1998. During summer and fall 1998 further enhancements have been made to both performance and functionality. Espresso’s most recent version in v0.3 For a list of missing or incomplete features in Espresso along with suggestions for optimizations and improvements see “Proposals for future work” on page 31.
1.3 Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
<th>Online reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>JLS</td>
<td>Java language specification</td>
<td><a href="http://java.sun.com/docs/books/jls/index.html">http://java.sun.com/docs/books/jls/index.html</a></td>
</tr>
<tr>
<td>JIM</td>
<td>JavaImportManager</td>
<td>none</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus Naur form</td>
<td>none</td>
</tr>
<tr>
<td>RHS</td>
<td>right-hand-side of a production</td>
<td>none</td>
</tr>
<tr>
<td>LHS</td>
<td>left-hand-side of production</td>
<td>none</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract syntax tree</td>
<td>none</td>
</tr>
</tbody>
</table>

1.4 Conventions

Class names, Java language statement and expression names are written in italic.

2.0 Architecture

Espresso is built using a compiler construction tool called JavaCC from SunSoft (trademark Sun Microsystems). Given a JavaCC grammar file as input, JavaCC constructs a LL(k) parser along with a token manager implementing the lexer for the language to be compiled.

The JavaCC grammar file for Espresso (Java1.0.2.jj) contains the following building blocks:

- The lexical definitions using regular expressions
- The productions for the Java grammar using a BNF like notation. Embedded in these productions are semantic actions to direct the process of parsing and to construct the abstract syntax tree.
- A block of Java code which represents the skeleton for the parser class which is constructed.

Each RHS of a grammar productions is turned into a method of the constructed Java parser, and each LHS is turned into an appropriate method call.

The generated Java parser is used to parse the Java source code. During this compilation step, an abstract syntax tree is constructed which forms the basis for the type checking and code generation phase.

A complete set of documentation for JavaCC is available through the JavaCC homepage http://www.metamata.com/javacc/DOC. The Espresso home page also contains some introductory slides to basic concepts of JavaCC.
The package *JavaClass* used to read and write Java class files, and the package *Class-Gen* used during code generation were downloaded from Markus Dahm home page at http://www.inf.fu-berlin.de/~dahm/JavaClass/ and adapted to our needs.

### 2.1 Overview

The UML class diagram in *Figure 1 on page 6* shows the top level structure of Espresso, including the root of the abstract syntax tree.

**FIGURE 1.** UML Class Diagram with class Espresso as top level class

Note that an object of type *CompilationUnitNode* represents the root of the AST.
2.2 The main class Espresso

*Espresso* is the main class of the Espresso compiler. Its `execute` method creates an object for each, the `SymbolTable`, the `JavaParser` and the `JavaImportManager`.

The UML sequence diagram in Figure 2 on page 7 shows an overall picture of the order, in which Espresso goes through the different phases of compiling.

First the initial environment is set up. Next the `parseProgram` method of `JavaParser` is called which returns the root of the AST that was constructed during this step.

On the root node of the AST (an object of type `CompilationUnitNode`), Espresso invokes the `typeCheck` method. Each node propagates the `typeCheck` call down the AST according to its own type checking logic.

If no error occurred during the type checking phase, Espresso calls the `translate` method on the root node of the AST which triggers the generation of code for the Java Virtual Machine JVM. Again, each node propagates the `translate` call down the AST if required by its translation rules.
2.3 The abstract syntax tree (AST)

During the parsing step of Espresso, an AST is constructed. The nodes of the AST are instances of classes which are part of a large class hierarchy with the class `SyntaxTreeNode` as its common base class. All AST nodes dealing with the various types of expressions share the common base class `ExpressionNode`. Similarly, nodes representing the different types of statements share the common base class `StatementNode`.

A detailed documentation of the class hierarchy is available through the Espresso home page (see javadoc). A graphical representation of the class hierarchy is available through the SGI Java Development Environment CosmoCode.

With the nodes of the AST being related in a class hierarchy, methods for type checking, evaluating expressions and translating can be defined as abstract methods at the appropriate levels, and concrete implementations can be provided. Using this approach, common functionality of related classes can be factored out and appropriate code can be placed in common ancestors in the class hierarchy.

2.4 The symbol table

Espresso’s symbol table is implemented as a flat data structure using a hashtable. In order to deal with the different scopes, a hierarchical key concept was developed. The key entries in the symbol table are represented by objects of type `Symbol`, a class that provides methods to construct symbols for the different entities, and to retrieve their parts.

*Table 1 on page 8* shows the entities currently maintained in the symbol table, along with the symbol encoding scheme for each entity.

<table>
<thead>
<tr>
<th>Symbol type</th>
<th>Encoding Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages</td>
<td>package name</td>
</tr>
<tr>
<td>Types (Classes and Interfaces)</td>
<td>package.name.class name</td>
</tr>
<tr>
<td>Fields</td>
<td>package.name.class name.field name</td>
</tr>
<tr>
<td>Methods</td>
<td>package.name.class name.method name.(l)</td>
</tr>
<tr>
<td>Formal Parameters</td>
<td>package.name.class name.method name.0.formal parameter name</td>
</tr>
<tr>
<td>Local Variables</td>
<td>package.name.class name.method name.scopeid.local variable name</td>
</tr>
<tr>
<td>Labels</td>
<td>label name:</td>
</tr>
<tr>
<td>Primitive Operators</td>
<td>symbol for operators. <em>Examples</em>: “+”, “==”...</td>
</tr>
</tbody>
</table>

For each of this entities, the symbol table provides an `add` and a `lookup` method. The reason for having entities dealing with *primitive operators* will be discussed in *Section 4.3 on page 21*. 
2.4.1 Relating symbol table and AST

During the parsing phase of Espresso, appropriate entities are added to the symbol table. This includes entities appearing in the compiled unit as well as relevant entities (type declarations, method declaration, field declarations) which needed to be imported from previously compiled Java class files.

Instead of providing new classes to contain relevant information associated with each type of entity in the symbol table, each entry points back to the node(s) in the AST where it was created, or where relevant data associated with it can be retrieved.

*Table 2 on page 9* shows the different entities of the symbol table and the objects that are stored along with them.

<table>
<thead>
<tr>
<th>Symbol type</th>
<th>Related AST nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages</td>
<td><em>CompilationUnitNode</em></td>
</tr>
<tr>
<td>Types (Classes and Interfaces)</td>
<td><em>TypeDeclarationNode</em></td>
</tr>
<tr>
<td>Fields</td>
<td><em>FieldDeclarationNode</em></td>
</tr>
<tr>
<td>Methods</td>
<td><em>Vector</em> of <em>MethodDeclarationNode</em></td>
</tr>
<tr>
<td>Formal Parameters</td>
<td><em>FormalParameterNode</em></td>
</tr>
<tr>
<td>Local Variables</td>
<td><em>LocalVariableDeclarationNode</em></td>
</tr>
<tr>
<td>Labels</td>
<td><em>LabeledStatementNode</em></td>
</tr>
<tr>
<td>Primitive Operators</td>
<td><em>Vector</em> of <em>MethodType</em></td>
</tr>
</tbody>
</table>

a. Overloading of methods
b. Overloading of primitive operators

The approach of maintaining imported entities in the same manner as entities present in the compiled unit allows for a uniform handling during type checking and code generation.

*Figure 3 on page 10* shows a small compilation unit along with the resulting AST and with the entries created in the symbol table.
FIGURE 3. Compilation unit with AST and symbol table

```java
package Test;
class Hello {
    public void helloYou(String you) {
        System.out.println("Hello" + you);
    }
}
```

Note, that for entries of the symbol table created by the JavaImportManager (marked JIM in rightmost column), appropriate nodes for the AST are also created but not shown. The AST nodes marked 6 to 9 have triggered loading of classes java.lang.System and java.io.PrintStream and as a result entries 6 to 9 were added to the symbol table.
2.5 JavaImportManager JIM

JIM provides a set of methods to accomplish the following two tasks during compilation of a unit:

- Determine whether a simple name appearing in the source code can unambiguously be fully qualified as a type name.
- Make appropriate portions of compiled Java class files available to the environment of Espresso.

Each Java class file contains compiled code for exactly one class or interface. The exact format of a Java class file is part of a standard defined by Sun Microsystems and is available online at [http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html](http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html). This standard guarantees that compiled Java classes are portable across platforms and operating systems.

Alternatively, Java class files may be grouped together in Zip files, or starting with JDK 1.1.4 in Jar files, a new archive format promoted by Sun.

JIM fully supports all currently defined formats. To accomplish the rather tedious task of reading the content of class files and of constructing the corresponding constant pool, JIM uses a package called classgen which is available to the public under the GNU licence agreement.

2.5.1 The Java CLASSPATH

The Java Virtual Machine JVM runtime system provides access to the Java resource CLASSPATH.

This resource is a vector of locations in the host file system at which Java class files can be lookup up. This locations are either root directories of packages or path descriptions to class files which are grouped together in either Zip files or Jar files.

Part of the content of the CLASSPATH vector is predefined by resource definitions set up by the JDK installation procedure. Additional definitions can be added by the user through the CLASSPATH environment variable.

2.5.2 Fully qualifying names

The Java Language Specification JLS defines a mode of importing classes called import on demand (Ex. java.lang.*).

Instead of explicitly listing each class in the import section of the java source file, import on demand can be used.

If during compilation, a simple name possibly representing a type name is encountered, JIM has to determine whether this simple name can be matched with a Java class file name. To accomplish this task it has to build the cross product between the entries in the CLASSPATH vector and the entries in the import on demand list, then patch the simple name to each of them and try to look it up. If a lookup was successful, the concatenation of the simple name and the appropriate portion of the import on demand entry repre-
sents the fully qualified name for the type. In addition, the JLS requires that a simple
name never refers to a type name that is present in more than one package. JIM makes
sure that a simple name can be qualified uniquely and issues an error otherwise.

To improve the lookup performance, JIM uses several caches to register misses as well
as hits.

2.5.3 Loading class files
Portions of Java class files imported by the compiled unit need to be added to the envi-
ronment of Espresso. For each imported class, JIM adds the appropriate type to the sym-
bol table.

For each non private method found in the class file, JIM adds a method to the symbol
table. Similarly, for each non private field, a field is added to the symbol table.

For each symbol added to the symbol table, an appropriate AST node is stored along
with it.

The loading of a class file is triggered by the following situations (provided the class is
not itself part of the compiled unit):
- a class is imported by default as defined by the JLS.
- a class is explicitly mentioned in the import section of the compiled unit.
- a class/interface is the superclass/superinterface of the class/interface that is com-
piled.
- a class is an interface, that is implemented by a class in the compiled unit.
- a class was imported on demand.
- if during type checking, a class is needed either to determine the most specific
  method to be invoked by a method expression or to verify access modifiers and
types of fields appearing in field expressions.

3.0 Parsing and constructing the AST

Espresso’s parser JavaParser is constructed by JavaCC from the JavaCC grammar file
java1.0.2.jj, which contains the code specifying the lexical definitions and the grammar
productions for the language. Grammar productions are augmented by semantic actions
to manage and execute the construction of the AST and to control and update the envi-
ronment as parsing goes along.

General concepts of LL(k) parsing are not discussed here and are available through text-
books on compiler construction. For details on the JavaCC grammar and the general
environment of JavaCC read the mini tutorial on JavaCC available through the JavaCC
home page..

This section mainly elaborates on special cases during parsing, which are related to the
following fields of JavaParser:
- className_d
3.1 Keeping track of actual package, class, method

The class `JavaParser` provides a set of fields containing symbols which are used often during parsing. Those of particular importance for constructing symbols during parsing are presented here.

There is a field `className_d` which at any point during parsing contains the fully qualified name of the currently parsed type (class/interface). Because a Java compilation unit may contain the definition of more than one type, this field needs to be updated each time a new type declaration is seen.

The field `methodName_d` always holds the fully qualified name of the currently parsed method. It is used in mostly in places where symbols for local variables need to be constructed.

3.2 Local scopes

During parsing, the scoping rules for classes, methods, fields, formal parameters and local variables have to be enforced. For classes, methods and fields, the scheme using package name, class name and field-/method name suffices to provide for the full flexibility defined by the language.

For names of formal parameter and local variables, an additional scope identifier is needed.

Espresso solution to this problem is to provide a scope stack, a data structure which is maintained in the `JavaParser` field `scopes_d`. The scope stack is implemented as a stack containing instances of `Integer` objects.

Each time a new local scope is entered, a fresh object of type integer is retrieved from `JavaParser` field `counter_d` and pushed onto the scope stack.

Names of formal parameters and local variables appearing within a particular scope receive the current top of the scope stack as their scope identifier. This identifier along with the fully qualified method name and the name of the variable itself suffices to uniquely identify a local variable or a formal parameter.

Whenever the parser sees the end of a scope, the current top of the scope stack is popped off.
It might be worth pointing out, that names of formal parameters and local variables
defined at the outermost scope of a method share the same scope.

Similarly, variables declared in the initializer part of the \textit{for} statement share the same
scope with variables declared at the outermost level within the \textit{for} statement.

3.3 Wrapping of control statements

In order to ease code generation and to avoid passing around break lists and continue
lists (containing branch instructions to be backpatched) across statement and block
boundaries, a number of control statements are wrapped with labeled statements during
parsing. The list of control statements and expression lists currently wrapped by the
parser includes those, which either potentially contain \textit{break} and/or \textit{continue} statements,
or for which a special handling for the \textit{continue} statement is needed.

The language constructs currently handled this way are:

- \textit{switch} statement
- \textit{while} statement
- \textit{do} statement
- \textit{for} statement
- The expression list representing the \textit{for update} clause of the \textit{for} statement.

If the statements listed above are not already labeled in the source program, the parser
pushes a fresh label onto the label stack maintained in \textit{JavaParser} field \textit{label_d} and
wraps the statement with a labeled statement using this newly created label. For the
update clause of the \textit{for} statement, a new label is always created. If during parsing of the
above constructs, a \textit{break} or \textit{continue} statement is seen and the optional label associated
with the statements is not present in the source program, than the parser assigns the cur-
tent top of the label stack as the label to it.

Whenever the parser reaches the end of one of the statements above, the current top of
the label stack is popped off.

Using this method, \textit{break} and \textit{continue} statements are always related to the proper sur-
rounding statements, regardless of whether the source program contains a label or not.

The method used by the parser to construct unique labels is identical to the one used for
generating unique scope identifiers.

3.4 Adding a default constructor

The Java language does not enforce the programmer to provide a constructor for a class,
but instead requires the compiler to add a default constructor, if the source program does
not contain one.

Therefore, whenever the end of a class declaration is reached, the parser has to verify
whether at least one constructor was defined, and add a default constructor to the class
body otherwise. This default constructor needs to call of the default constructor of the superclass of the class.

3.5 Non static initializers for fields

Non static fields of a class (instance variables) are initialized by the JVM with a default value whenever a new object of this type is created. If the source program defines an initializer for the field, this initializers has to be used instead. The solution proposed by the JLS is to add all initializers in textual order to each constructor, by turning them into assignments to the appropriate fields.

To implement this solution in Espresso, JavaParser maintains a special field fieldInit_d representing a block, to which field initializers are added whenever they are encountered, and after constructing an assignment expression for it.

After verifying for the presence of at least one constructor in the class (at the end of ClassDeclaration()), the block containing these initializers is added to each constructor found in the body of the class.

3.6 Static initializers, and initializers for static fields

Static fields of a class are initialized exactly once, when a class is loaded by the JVM.

If the programmer provides initializers for static fields, or if special static initializer blocks are defined, then a special static constructor <clinit> needs to be added to the class body by the compiler.

The JLS requires, that these initializers are added to the static constructor, and that they are executed in the textual order of there appearance in the source program.

Whenever the parser encounters a declaration of a static field with an initializer, it turns the initializer into an assignment to that field, and adds it to the JavaParser field staticInit_d. In a similar way, statement expressions found in static initializer blocks are added to the same field, always taking the textual order into account.

At the end of parsing a class declaration, the parser adds a static constructor to the class body (provided staticInit_d is not empty). and adds statements and statement expressions contained in static_init_d to the block of the static constructor.

3.7 Verifying closure of the compiled unit

As a last step of the parsing phase, we need to verify whether all references to types (classes/interfaces), methods and fields appearing in the source code have been uniquely matched with either declarations found in the compiled unit itself or with classes loaded by JIM.

The step is needed, because the Java language allows forward references to types, methods and fields.

Resolving forward references to fields and methods is done during parsing.
If type names appearing in type expressions, this expressions and super expressions were not resolvable during parsing, the corresponding symbols are added to a special JavaParser field forwardTypes_d.

This last step has to verify, whether each type contained in this field is known in the environment by now, or reports an error otherwise.

4.0 Type checking

After an AST was successfully constructed by JavaParser, type checking is performed on the nodes of the AST. Each node of the AST provides a typeCheck method that is responsible to perform the type checking of the language construct is represents and to propagate type checking do its successors 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 to encode these types according to the encoding scheme defined by the JVM to make them comparable to the ones loaded from class files.

The main tasks of type checking is to determine the resulting types of expressions, to verify, that access restriction for packages, classes, methods and fields are respected, and in general, to enforce the typing rules imposed by the JLS.

This section introduces to the Espresso’s type checking concepts and highlights some of the more difficult or non standard cases.

4.1 Espresso’s type hierarchy

The type hierarchy developed for Espresso is among 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 the special type boolean, and all the numeric types.

Numeric types are further divided into integral types (which includes the types byte, short, int, long and char), and floating point types (float, 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.

For a comprehensive introduction to Java's type system see JLS Ch. 4.1

Espresso’s type hierarchy reflects in large the hierarchy presented above. The differences are the following:
Type checking

- The `null` type is part of the `reference` types
- The type `void` is included with the `primitive` types

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

The common base class of Espresso’s type hierarchy is the class `Type`. It defines a number of abstract binary predicates and a distance function.

The UML class diagram in Figure 4 on page 17 shows Espresso’s type hierarchy in detail.

The remainder of the section provides an exact definition of the predicates shown in the above class diagram.

The definition of the `distanceTo` function will be given in the subsection covering the distance matrix.
4.1.1 identicalTo
A primitive type is identical to another type, if the other type represents the same primitive type.

A class type is identical to another type, if the other type is a class type with the same name.

An array type is identical to another type, if the other type is an array type with the same dimensions, and if the two base types are identical to each other.

The null type is identical to another type, if the other type is also the null type.

A method type is identical to another type, if the other type is also a method type, and moreover, if the return types, and the types of the formal parameters are identical to each other. (There is also a version that ignores the return type: equality modulo return type).

4.1.2 subTypeOf
A primitive type is subtype of another type, if the two types are identical to each other.

A class type is a subtype of another type in the following cases:
- the two type are identical
- the supertype of the class type is a subtype of the other.
- if the class type extends or implements interfaces, and one of the interfaces is a subtype of the other type.

A class type is not a subtype of the other, if they are not identical and the class type itself is Object.

The null type is subtype of another type, if the other type is either identical to the null type, or the other type is a class type or an array type.

An array type is a subtype of another type in the following cases.
- the two types are identical
- the other type is also an array type with identical dimensions, and the base type of the array type is a subtype of the base type of the other array type.
- the other type is a class type of type Object or Cloneable.

A method type is a subtype of another type, if the two are identical.

4.1.3 superTypeOf
A primitive type is supertype of another type, if the two types are identical.

A class type is a supertype of the other type, if the other is a subtype of it.

The null type is a supertype of another type, if the other is also the null type.
An array type is a supertype of another type, if the other type is a subtype of it.

A method type is a supertype of another type, if the two are identical.

### 4.1.4 relatedTo
Two types are related to each other, if they are in subtype relation.

### 4.2 The distance metric

The typing rules imposed by JLS require a compiler to choose the most specific method among all the methods that are applicable for a particular method invocation. Moreover, the JLS also defines a host of conventions a compiler has to deal with, when type checking of expressions involving primitive operators is performed. This includes, which primitive types are assignable to each other, and what the result type of a unary or a binary primitive operator is, taking the types of the operands into consideration.

In the course of developing Espresso, Santiago M. Pericas discovered a central metric that allowed to cover the different requirements with a single concept, a distance function. The general idea is to define quantities bigger than or equal to zero for operations which are allowed between certain types, and to define negative quantities for operations which are not applicable between certain pairs of types.

During checking, the smallest quantity bigger than or equal to 0 is the one that needs to be chosen by the type checker. If no such smallest positive quantity was found for a particular case, then that case was not typeable.

The matrix in *Table 4 on page 19* documents the distanceTo function for all types of the type hierarchy.

<table>
<thead>
<tr>
<th></th>
<th>boolean</th>
<th>byte</th>
<th>short</th>
<th>char</th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
<th>void</th>
<th>Null</th>
<th>Class</th>
<th>Array</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>0</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
</tr>
<tr>
<td>byte</td>
<td>-oo</td>
<td>0</td>
<td>1</td>
<td>-oo</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
</tr>
<tr>
<td>short</td>
<td>-oo</td>
<td>-oo</td>
<td>0</td>
<td>-oo</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
</tr>
<tr>
<td>char</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
</tr>
<tr>
<td>int</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
</tr>
<tr>
<td>long</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
</tr>
<tr>
<td>float</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>0</td>
<td>1</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
</tr>
<tr>
<td>double</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>0</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
<td>-oo</td>
</tr>
</tbody>
</table>
Type checking

The distance between class types (indicated by $C$ in the matrix above) is calculated as follows:

- The distance between two identical class types is 0.
- The distance between $Object$ and any other class type is $-\infty$.
- If the class type is a subtype of the other type, than the distance is equal to the number of levels of subtypes between them. The direct supertype is at distance 1.
- 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.

The distance between array types of identical dimensions (indicated by $A$ in the above matrix) is defined as follows:

- If the base type of both types are class types, then the distance between an array type and another array type is equal to the distance of the base type of the array to the base type of the other type.
- If not both base types are class types, than the distance between the two array types is 0, if the base types are identical.
- In all other cases, the distance is $-\infty$.

The distance between two method types (indicated by $M$ in the matrix above) is defined as the sum of the distances between the individual arguments of the two methods. More formally, let $S = (s_1 x \ldots x s_n)$ and $T = (t_1 x \ldots x t_n)$ be two method types and let $\delta$ be the distance function. Then the distance $\delta (S, T)$ is defined as:

$$\sum_{i=1}^{n} \delta(s_i, t_i)$$

If any single distance between an argument of the method type $S$ and the corresponding argument of the method type $T$ is less than 0, the resulting distance is $-\infty$. 

### Table 4.

Distance matrix

<table>
<thead>
<tr>
<th></th>
<th>boolean</th>
<th>byte</th>
<th>short</th>
<th>char</th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
<th>void</th>
<th>Null</th>
<th>Class</th>
<th>Array</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>Null</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>0</td>
<td>0</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>Class</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$C$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>Array</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$A$</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>Method</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$M$</td>
</tr>
</tbody>
</table>
4.3 Type checking for primitive operators on primitive types

The JLS defines pages and pages of rules that need to be enforced when applying primitive operators on primitive types, including conditions, under which widening or narrowing of primitive types has to take place.

Espresso’s type checking implements all this rules in a uniform and very elegant way. For all the unary and binary operators on primitive types, an initial environment is added to the symbol table. This environment defines for each unary primitive operator, for which primitive type it is defined, and what the resulting primitive type for the operation is. This is done by adding the primitive operator to the symbol table, along with a method type for each possible type supported by the operator describing the type of the argument and the result type.

Binary operators on primitive types are defined in a similar way by providing a method type with two arguments and a result type.

Note that the order in which these method types are added to the symbol table for a particular operator directly affects the desired outcome of a lookup.

To type check an expression involving a primitive operator we only need to determine the types of its arguments, create an appropriate method type, and use the definitions given in the initial environment together with the $\text{distanceTo}$ function presented above, to determine whether the expression type checks and what the result type is.

4.4 Finding the most specific method

Espresso’s implementation of the type for class types ($\text{ClassType}$) provides a method $\text{findMethod}$, to determine the most specific method declaration for a particular method invocation expression ($\text{MethodExpNode}$) as defined in JLS 15.11.2.2

Type checking of $\text{MethodExpNode}$ first sets up an instance of class $\text{MethodDesc}$, which contains a method type describing the signature of the method invocation expression, the name of the method and some other fields that are used to keep track of additional information that is collected during the search.

The overall idea is to pass this information to the initial class type determined by the parser, and to get the most specific method back.

Within $\text{findMethod}$, the symbol table is looked up for matching methods using the name of the type and the method name that was initially provided. Methods obtained from the symbol table with a matching number of arguments are sorted out, and among those, the most specific is chosen by applying the $\text{distanceTo}$ function of the initial method type to the type of each candidate method. Once the most specific candidate method declaration of a particular type (class/interface) is determined, the search continues recursively up to that types supertype.

The search either stops when an ambiguity among candidate method declarations was encountered (throws a type check error), or when Java’s primordial type $\text{Object}$ was reached. If searching was successful, the most specific method is returned to the initial caller.
During the search up the type inheritance 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.

The same search idea is also applied, when determining the most specific constructor declaration while type checking constructor invocation expressions and allocation expressions.

### 4.5 Finding the right field

Espresso’s `ClassType` also provides a method for finding fields in order to type check field expressions. Unlike for finding methods, finding the appropriate field declaration not only needs to check in superclasses of a particular type, but in its superinterfaces as well. The technique applied for this search is similar to the one used for finding methods.

### 4.6 Type checking of initializers for variables

The JLS defines special rules, that apply to initializers of variables. While in normal assignments an integer constant is always of type int, and the programmer needs to explicitly cast the type down when the RHS of the assignment is an integral type smaller than int, initializers of variables need to be treated differently.

In fact, the JLS requires, that initializers of variables are always evaluated. If the result of the evaluation is a constant expression, then the compiler needs to verify, whether the value of the constant expression fits into the primitive type that needs to be initialized.

The expression hierarchy of the AST provides a special evaluation function for each kind of expression which possibly evaluates to a constant expression. The expression base class `ExpressionNode` defines a method `evaluate`. Each subclass of `ExpressionNode` that potentially evaluates to a constant expression overwrites this method and invokes the `evaluate` method of its arguments. If no exception was thrown, the appropriate operation is performed (Ex: addition or subtraction in `AndExpNode`) on the constant expressions resulting from its arguments and handed up the expression tree in the AST. The `ExpressionNode` then provides a second method `evaluateExp` that can be called in the type checking phase, which either catches an exception and returns `null` indicating, that the initializer did not evaluate to a constant expression, or returns the evaluated value, which is used by the type checker to verify, whether it fits into the type of the initialized variable.

### 5.0 Code generation

After successful completion of type checking, the last phase of the compiler is responsible for generating code.

The main class Espresso initiates the translation phase by invoking the `translate` method of the root node of the AST. The different nodes in the AST propagate the translation calls down to their children as needed.
Espresso uses the package *classgen*, which is available under GNU's licence agreement. 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.

This section first introduces to the important classes and methods of the *classgen* package. Then it presents the translation step in the *TypeDeclarationNode*, which initiates the translation of methods and is responsible for adding them to their Constant Pool.

Further topics include Espresso’s solution for lazy evaluation of boolean expressions, and the scheme that is used for back patching, as well the translation of break- and continue statement.

### 5.1 The overall translation picture

Generating code for a 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.

To construct an instance of *ClassGen*, the name of the Java source file, the type name, the name of the supertype, the types access flags and the list of interfaces that are extended or implemented by the type need to be supplied.

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 to translate the Java code contained in the method body, and is discussed in more detail in this section.

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.

### 5.2 Translating a method body

For each method declaration with a non empty body, an instance of *classgen.MethodGen* needs to be created. We need to supply the methods access flags, the encoding of the return type and the types of the arguments together with the argument names (*become local variables*), the class and method name, and finally an instance of *classgen.InstructionList* to hold the byte code instruction stream.

Among the methods available in class *MethodGen*, Espresso mainly uses those for retrieving the methods instruction list and for dealing with *local variables*.

For nodes of the AST representing expressions or statements, the *translate* method has a uniform signature, taking the current java class and the currently translated method as parameters. This two objects suffice for most cases to provide access to the java class and to the code that was generated so far.
Espresso’s solution for dealing with those cases, where access across statement or expression boundaries is needed is discussed in more detail in the following subsections.

After code generation for the method body is completed, NOP’s are removed from the instruction list, and the translated method is added to the java class. Because of the way the package classgen is constructed, this needs to be done as the last step in translating a method body.

5.3 The role of Espresso’s type hierarchy

Beyond type checking, Espresso’s type hierarchy is also heavily used during translation. The JVM instruction set distinguishes operand types by providing distinct opcodes for operations on its various data types. The first distinction is made between reference types (class type, array type, null type) and primitive types. Looking in more detail at the instruction set of the JVM, the following instruction families can be made out:

- class type
- array type
- null type
- primitive type int
- primitive type long
- primitive type double

The primitive type boolean is always treated as type int. For most operations on the Java types byte, char and short, no special instructions are available. Instead a set of instructions is provided to convert forth and back between those types and the type int.

In order to minimize data type specific case analysis in the nodes of the AST during translation, Espresso’s type hierarchy provides a set of methods to construct appropriate instructions depending on the type of the operand. Instructions constructed this way include:

- load and store instructions
- all primitive operations
- instructions for comparison and branching
- load and store array elements
- array creation
- cast instructions for narrowing and widening
- return instruction

To illustrate the principle, consider the case of translating the return statement. Espresso’s top type Type declares an abstract method RETURN. Table 5 on page 25 shows the different types of return instructions which need to be created, along with the subclass of Type which provides the appropriate implementation:
5.4 Controlling translation for lvalues and rvalues

One of the problems that needs to be solved when building a compiler for an imperative language is to make sure, that operations that change the state of variables are performed only on variables that represent updateable locations in memory. In the grammar used for Espresso, these updateable locations are represented by instances of the following classes of the AST.

- FieldExpNode
- FormalExpNode
- LocalExpNode
- ArrayExpNode

During the typechecking phase, the predicate ExpressionNode.leftValue() is used to verify, whether a updating operation is performed on an appropriate type of expression.

Generating code for these expressions requires to translate an expression according to its use as either a right value rvalue (read current content of location) or as a left value lvalue (write to location).

To control the translation of such an expression, the base class of all expressions ExpressionNode provides a flag leftValue_d which is turned on or off either already in the parsing phase, or when appropriate, later in the translation phase.

A number of expressions require, that code needs to be generated for a lvalue and for a rvalue. Among these expressions are field expressions in pre-/postIncrement expressions and pre-/postDecrement expressions, depending on the context of their use.
The following example illustrates Espresso’s translation for a case, where \textit{rvalue} and \textit{lvalue} translation is needed (flag \textit{leftValue.d} is turned on and off)

\begin{verbatim}
class Demo {
    boolean b;
    int u, v;
    public void preDecPostInc( ) {
        b = --u==v++;  
    }
}
\end{verbatim}

Figure 5 on page 26 shows the relevant part of the AST, that results from parsing the code fragment containing the assignment \( b = --u==v++ \)

Because the JVM is a stack based machine, translating code for it corresponds to a \textit{post order} traversal of the AST. This is reflected by the numbering of the nodes above We will use the resulting code shown in Table 6 on page 27, to walk through the translation.
TABLE 6. JVM code resulting from translation of the AST for Method void preDecPostInc()

<table>
<thead>
<tr>
<th>Code offset</th>
<th>JVM instruction</th>
<th>Step</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>aload_0</code></td>
<td>1</td>
<td>Since FieldExpNode for b is used as an lvalue here, no load is required, we prepare just for store</td>
</tr>
<tr>
<td>1</td>
<td><code>aload_0</code></td>
<td>3</td>
<td>FieldExpNode for u is translated as lvalue first, we prepare for store, and step 4 does not generate load instruction</td>
</tr>
<tr>
<td>2</td>
<td><code>aload_0</code></td>
<td>3</td>
<td>Now FieldExpNode for u is translated as rvalue and a load instruction is generated</td>
</tr>
<tr>
<td>3</td>
<td><code>getfield #15 &lt;Field int u&gt;</code></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>iconst_1</code></td>
<td>5</td>
<td>This is the translation for the predecrement</td>
</tr>
<tr>
<td>7</td>
<td><code>isub</code></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><code>dup_x1</code></td>
<td>5</td>
<td>We need to duplicate the current top of stack, because we need it for the comparison</td>
</tr>
<tr>
<td>9</td>
<td><code>putfield #15 &lt;Field int u&gt;</code></td>
<td>5</td>
<td>Store predecremented value to field u. Here we use up the reference created at code offset 1 during lvalue translation</td>
</tr>
<tr>
<td>12</td>
<td><code>aload_0</code></td>
<td>6</td>
<td>FieldExpNode for v is translated as lvalue first, we prepare for store, and step 7 does not generate load instruction</td>
</tr>
<tr>
<td>13</td>
<td><code>aload_0</code></td>
<td>6</td>
<td>Now FieldExpNode for v is translated as rvalue and a load instruction is generated</td>
</tr>
<tr>
<td>14</td>
<td><code>getfield #17 &lt;Field int v&gt;</code></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td><code>dup_x1</code></td>
<td>8</td>
<td>We need to duplicate the unmodified value of v, because comparison is performed on original value</td>
</tr>
<tr>
<td>18</td>
<td><code>iconst_1</code></td>
<td>8</td>
<td>This is the code for post increment</td>
</tr>
<tr>
<td>19</td>
<td><code>iadd</code></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td><code>putfield #17 &lt;Field int v&gt;</code></td>
<td>8</td>
<td>Store postincremented value to field v. Here we use up the reference created at code offset 12 during lvalue translation</td>
</tr>
<tr>
<td>23</td>
<td><code>if_icmpne 30</code></td>
<td>9</td>
<td>Translation of comparison == (check next section for reason of !=)</td>
</tr>
</tbody>
</table>
5.5 Lazy evaluation of boolean expressions, backpatching and synthesizing

Starting with Espresso v0.3 the translation of boolean expressions first tries to evaluate boolean expressions to obtain a constant expression. If not applicable, lazy evaluation of boolean expressions needs to be performed. To provide the necessary infrastructure for backpatching, the AST expression base class ExpressionNode defines a field trueList_d (for the true cases) to hold branch instructions created during translation, which need to be backpatched when the target of the branch instruction is known. Similarly, a field falseList_d is provided for the false case. As translation of boolean expressions proceeds, trueList_d and falseList_d are passed up the expression tree and either merged or swapped according to the semantic of the particular expression. In most cases, backpatching is performed in the course of translating the enclosing control statement. Backpatching also needs to be performed, when boolean expressions need to be synthesized. This has to be done for instance in the following situations:

- A boolean expression is assigned to a boolean variable
- A boolean expression is passed as a parameter to a method call, a constructor invocation or an allocation expression
- A boolean expression is part of an additive expression which represents a string concatenation.

In order to synthesize boolean expressions, the class ExpressionNode provides a method translateSynthesized that performs the required task. Subclasses of ExpressionNode override this method if they require a more specific implementation.

In order to produce more compact code, Espresso translates boolean operators in a not obvious manner. All boolean operators found in the source code are translated to their negation. The true case always falls through, and the false case produced is added to the falseList to be backpatched later. The instruction at offset 23 in Table 6 on page 27 is an example of this strategy, where an equality expression produces an if_icmmpne.

Given this setup, translateSynthesized basically performs a regular translate on its parts, and then adds the appropriate code to leave the constant one (iconst_1) on the operand stack and appends an unconditional branch right after to the instruction stream. This
instruction handle is returned to the caller of the synthesized translation at the end. Then it append\s the constant zero (inconst\_0) to the instruction stream, and backpatches the branch instructions contained in the falselist to this location.

We will use the following code fragment and parts of the resulting AST shown in Figure 6 on page 29 to illustrate this principle in more detail. The example shows lazy evaluation of boolean expression, Espresso’s principle of translating boolean operators to their negation, and the process of synthesizing a boolean expression for the purpose of performing an assignment.

```java
class Demo {
    boolean b;
    int u, v, w;
    public void demoLazy( ) {
        b = u!=v and v==w ;
    }
}
```

---

**FIGURE 6.** Partial AST for example on lazy evaluation of boolean expressions
Table 7 on page 30 shows the code that was generated for the example above, along with remarks to particular translation steps.

### TABLE 7.

<table>
<thead>
<tr>
<th>Code offset</th>
<th>JVM instruction</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>aload_0</td>
<td>rvalue translation for LHS of assignment</td>
</tr>
<tr>
<td>1</td>
<td>aload_0</td>
<td>rvalue translation of e11</td>
</tr>
<tr>
<td>2</td>
<td>getfield #16 &lt;Field int u&gt;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>aload_0</td>
<td>rvalue translation of e12</td>
</tr>
<tr>
<td>6</td>
<td>getfield #18 &lt;Field int v&gt;</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>if_icmpeq 27</td>
<td>Translation of != in e1. Note that original comparison was replaced by its negation. The branch instruction is added to falseList of e1. If condition is false, we don’t need to evaluate e2 (lazy evaluation) True case falls through</td>
</tr>
<tr>
<td>12</td>
<td>aload_0</td>
<td>rvalue translation of e21</td>
</tr>
<tr>
<td>13</td>
<td>getfield #18 &lt;Field int v&gt;</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>aload_0</td>
<td>rvalue translation of e22</td>
</tr>
<tr>
<td>17</td>
<td>getfield #20 &lt;Field int w&gt;</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>if_icmpne 27</td>
<td>Translation of == in e2 Again original operation is negated, and branch is added to falseList of e2</td>
</tr>
<tr>
<td>23</td>
<td>iconst_1</td>
<td>This is the synthesized translation in e. Since the true case falls through, we need to push one to the operand stack. Then we need to add an unconditional branch to jump over the instruction which synthesizes the false case (This branch is returned to AssignmentNode to be backpatched, when the assignment instruction is added. Next we synthesize for the false case by pushing zero to the operand stack) and backpatch the merged falselists of e11 and e21 to this location.</td>
</tr>
<tr>
<td>27</td>
<td>iconst_0</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>putfield #14 &lt;Field boolean b&gt;</td>
<td>Translation of the assignment in AssignmentNode. Backpatch branch returned by translate-Synthesized to this location.</td>
</tr>
</tbody>
</table>
5.6 Translating continue statement

A continue statement may occur only within one of the iteration statement do, while and for. Control is transferred to the continue target, which is either the innermost enclosing iteration statement in case no continue label was provided by the programmer, or to the labeled iteration statement that matches with the label of the continue statement. The current iteration is immediately terminated, and the next one is started.

As was pointed out in “Wrapping of control statements” on page 14, Espresso makes sure, that each continue and break statement is labeled, and therefore matching with the appropriate continue target is relatively easy.

The translation of a continue statement needs to add an unconditional branch to the instruction stream. In order to avoid passing around the instruction which represents the continue target, continue branch instructions belonging to a particular continue target are collected during code generation in the field continuelist of the appropriate enclosing labeled statement (instance of LabeledStatementNode).

When translating a labeled statement, first a NOP instruction representing the most likely target of a potentially contained continue is added to the instruction stream. Then translation of the enclosed statement is performed. At the end, branch instructions added by enclosed continue statements to the continuelist need to be back patched with the continue target. In the general case, this target is the NOP instruction that was first created. If however the wrapped statement represents a for statement with an non empty update expression list, then the start of that expression list represents the correct continue target according to the semantic of the language. The labeled statement that encloses that expression list can be looked up in the symbol table using the label symbol that results from adding the string “for” to the label contained in labeled statement that wraps the for statement.

5.7 Translating break statement

Translation of the break statement works similar to translating the continue statement. The branch instruction created by the translation of the break statement are added to the breaklist of the appropriate labeled statement. After finishing translation of the enclosed statement in a labeled statement, a NOP instruction is added to the instruction stream, to serve as the target of the branch instructions contained in the breaklist.

6.0 Proposals for future work

This sections topic is to point out areas of future work.

It first presents a detailed list of language features of Java, that are currently not, or only partly implemented.

Next, some areas of possible optimizations are identified, with particular focus on generating more efficient code.
A last part suggests some fields of improvement concerning error handling and reporting.

6.1 Missing or incomplete features

This section identifies those parts of the Java language that are currently not implemented, or not fully available in Espresso.

In principle, Espresso in its current version conforms to the Java Language Specification JLS version 1.0.

6.1.1 Parsing

With the exception of some wired cases of forward references, the entire grammar of JLS 1.0 is supported by Espresso.

In some cases, Espresso simply overwrites access modifiers that are potentially present in the source program by setting default access modifiers instead (Example: field- and method declarations in interfaces) JLS Ch. 9.3 and 9.4.

The reason for this is, that the same grammar productions are currently used for classes and interfaces.

Final fields need to be sorted out in step that collects initializers of static variables.
### 6.1.2 Type Checking, Semantic Checks

#### TABLE 8. Topics to be completed in type checking and in semantic checks

<table>
<thead>
<tr>
<th>Main topic</th>
<th>Topic</th>
<th>Hints</th>
<th>Reference JLS</th>
<th>Status/Contributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class/Interface Declaration</td>
<td>Consistency among Class modifiers</td>
<td>not more than one of public/private/protected</td>
<td>JLS Ch. 8.1.2</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Class/Interface Declaration</td>
<td>Consistency among Class modifiers. Final classes</td>
<td>modifier pair final/abstract illegal</td>
<td>JLS Ch. 8.1.2.2</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Class/Interface Declaration</td>
<td>Consistency among Class modifiers. Final classes</td>
<td>Name of a class declared final may not appear in extents clause of the declaration of another class</td>
<td>JLS Ch. 8.1.2.2</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Method Declarations</td>
<td>Closure of non abstract classes</td>
<td>Closure of non abstract classes with respect to interface methods it promises to implement, and to abstract methods of superclasses</td>
<td>JLS</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Interface Declarations</td>
<td>Circular test for Interfaces</td>
<td>JLS 9.1.3</td>
<td></td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Field Declaration</td>
<td>Initializers for fields of interfaces</td>
<td>Check for presence of initializers</td>
<td>JLS Ch. 9.3.1</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Field Declaration</td>
<td>Consistency among field modifiers</td>
<td>not more than one of public/private/protected</td>
<td>JLS Ch. 8.3.1</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Field Declaration</td>
<td>Consistency among field modifiers</td>
<td>modifier pair final/volatile is illegal</td>
<td>JLS Ch. 8.3.1.4</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Field Declaration</td>
<td>Initializers of class (static) variables</td>
<td>Special rules apply</td>
<td>JLS 8.3.2.1</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Field Declaration</td>
<td>Initializers of instance variables</td>
<td>Special rules apply</td>
<td>JLS 8.3.2.2</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Field Declaration</td>
<td>Initializers of static variables</td>
<td>Check for abrupt completion by checked exception</td>
<td>JLS 8.3.2</td>
<td>Needs to be done</td>
</tr>
</tbody>
</table>
### TABLE 8. Topics to be completed in type checking and in semantic checks

<table>
<thead>
<tr>
<th>Main topic</th>
<th>Topic</th>
<th>Hints</th>
<th>Reference JLS</th>
<th>Status/Contributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Declaration</td>
<td>Initializers of static variables</td>
<td>Check for illegal reference of class variables declared in this class, whose declaration appear textually after the use.</td>
<td>JLS Ch. 8.5</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Interface Members</td>
<td>Initializers for fields</td>
<td>Initialization expression must not contain a reference by simple name to same field or to another field whose declaration occurs textually later in the same interface</td>
<td>JLS Ch. 9.3.1</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Array variables</td>
<td>Initializers of Arrays</td>
<td>90% missing</td>
<td>JLS Ch. 10.2</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Method Declaration</td>
<td>Consistency among method modifiers</td>
<td>not more than one of public/private/protected</td>
<td>JLS Ch. 8.4.3</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Method Declaration</td>
<td>Consistency among method modifiers</td>
<td>private/abstract, static/abstract, final/abstract, native/abstract pairs are illegal</td>
<td>JLS Ch. 8.4.3</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Class Declaration/Method Declaration</td>
<td>Consistency between class modifiers and method modifiers</td>
<td>Abstract method in non abstract class</td>
<td>JLS 8.4.3.1</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Method Declaration</td>
<td>Final methods</td>
<td>Check for overriding of final methods</td>
<td>JLS 8.4.3.3</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Method Declaration</td>
<td>Native methods</td>
<td>None</td>
<td>JLS 8.4.3.4</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Method Declaration</td>
<td>Method Throws</td>
<td>None</td>
<td>JLS 8.4.4</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Method Declaration</td>
<td>Hiding methods of superclasses and interfaces</td>
<td>Special checks needed</td>
<td>JLS Ch. 8.4.6</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Constructor Declaration</td>
<td>Consistency among constructor modifiers</td>
<td>not more than one of public/private/protected</td>
<td>JLS 8.6.3</td>
<td>Needs to be done</td>
</tr>
</tbody>
</table>
TABLE 8. Topics to be completed in type checking and in semantic checks

<table>
<thead>
<tr>
<th>Main topic</th>
<th>Topic</th>
<th>Hints</th>
<th>Reference JLS</th>
<th>Status/Contributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor Declaration</td>
<td>Default Constructor</td>
<td>Access modifier is taken from class, if class is declared public</td>
<td>JLS Ch. 8.6.7</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Constructor body</td>
<td>Explicit constructor invocation</td>
<td>Check for illegal mention of instance variables or instance methods declared in this class or any superclass, as well as for the use of this or super in any expression contained within an explicit constructor invocation statement.</td>
<td>JLS 8.6.5</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Statements/Expressions</td>
<td>Exception handling</td>
<td>TryStatement with associated Catch-Block or Finally-Block (100% missing)</td>
<td>JLS Ch. 11</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Statements/Expressions</td>
<td>Altering of final fields</td>
<td>Values of finals fields can never be altered, once they have been initialized. (Idea: extend definition of left Value)</td>
<td>JLS Ch. 8.3.1.2 JLS 15.14.2</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Statements/Expressions</td>
<td>Return statement</td>
<td>Using flow analysis, we need to check that in every possible path of execution, we have a return statement.</td>
<td>JLS Ch. 14.15</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Statements/Expressions</td>
<td>Throw Statement</td>
<td>100% missing</td>
<td>JLS Ch. 14.16</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Statements/Expressions</td>
<td>Try Statement</td>
<td>100% missing</td>
<td>JLS Ch. 14.18</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Statements/Expressions</td>
<td>Check for unreachable statements</td>
<td></td>
<td>JLS Ch. 14.19</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Statements/Expressions</td>
<td>Class Instance creation expression</td>
<td>Need to check, whether the class type mentioned in the expression is not an abstract class</td>
<td>JLS Ch. 15.8</td>
<td>Needs to be done</td>
</tr>
</tbody>
</table>
Proposals for future work

6.1.3 Code generation

<table>
<thead>
<tr>
<th>Topic</th>
<th>Hints</th>
<th>Reference JLS</th>
<th>Status/Contributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initializers for fields of interfaces</td>
<td>Need to be added to constant pool</td>
<td>JLS Ch.</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Array Initializers</td>
<td>100% missing</td>
<td>JLS Ch.</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>TryStatement/ThrowStatement</td>
<td>Exception Handling</td>
<td>JLS Ch. 14.16</td>
<td>Needs to be done</td>
</tr>
<tr>
<td></td>
<td>100% missing</td>
<td>JLS Ch. 14.18</td>
<td>Needs to be done</td>
</tr>
</tbody>
</table>
## Proposals for future work

### Table 9. Topics to be worked on in code generation

<table>
<thead>
<tr>
<th>Topic</th>
<th>Hints</th>
<th>Reference JLS</th>
<th>Status/Contributor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronized Statement</td>
<td>simple, need to add only <code>monitorenter</code> and <code>monitorexit</code></td>
<td>JLS Ch. 14.17</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>bitwise logical operators (&amp;, ^, !)</td>
<td>Maybe done</td>
<td>JLS Ch. 15.21</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Postfix Increment, Postfix Decrement</td>
<td>Does currently not generate proper code for integral types smaller than int</td>
<td>JLS Ch. 15.13.2, JLS Ch. 15.13.2</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Prefix Increment, Prefix Decrement</td>
<td>Does currently not generate proper code for integral types smaller than int</td>
<td>JLS Ch. 15.14.1, JLS Ch. 15.14.2</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Bitwise complement operator</td>
<td>100% missing</td>
<td>JLS Ch. 15.14.5</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>String concatenation operator</td>
<td>100% missing</td>
<td>JLS 15.17.1</td>
<td>Available in Espresso v0.3 Karl Doerig</td>
</tr>
<tr>
<td>Shift operators</td>
<td>100% missing</td>
<td>JLS Ch. 15.18</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Type comparison operator <code>instanceof</code></td>
<td>100% missing</td>
<td>JLS Ch. 15.19.2</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Compound assignment operators</td>
<td>100% missing</td>
<td>JLS Ch. 15.25.2</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Calculation of maximum stack size</td>
<td>Code attribute in the Java Class File. Is currently set to a fixed value</td>
<td>JVM</td>
<td>Needs to be done</td>
</tr>
<tr>
<td>Generation of LineNumber attribute</td>
<td>This is used only, to support debugging, and is thus not very critical</td>
<td>JVM</td>
<td>Needs to be done</td>
</tr>
</tbody>
</table>
6.2 Optimization

### TABLE 10. Topics for optimizations

<table>
<thead>
<tr>
<th>Main benefit</th>
<th>Hints</th>
<th>Status/Contributor</th>
</tr>
</thead>
</table>
| Reduce size of the generated JVM code | Removal of NOP instructions | Available in Espresso v0.3  
Santiago M. Pericas |
| Reduce size of generated code and improve execution time | Removal of goto’s to the next instruction | Available in Espresso v0.3  
Santiago M. Pericas |
| Reduce size of generated code and improve execution time | Replacing patterns of conditional branches, immediately followed by goto by the negation of the conditional branch, with the target of the goto as the branch target | Available in Espresso v0.3  
Santiago M. Pericas |
| Reduce size of generated code and improve execution time  
Improve time required for compiling | Consequent evaluation of expressions throughout the translation process, so that constant expressions get transformed into single load instructions. This can be done during the code generation phase, by providing and applying the evaluation function in all expressions. | Available in Espresso v0.3  
Santiago M. Pericas |
| Reduce size of generated code and improve execution time | Traditional optimizations based on flow analysis, like common sub expression elimination, code motion (place loop invariant outside loops) a.s.f | Needs to be done |

6.3 Improvements to error reporting

The following improvements should be considered or need to be done, concerning error reporting.

- We need to report location of syntax errors. Right now, no source code information is printed, when a syntax error occurs.
- Espresso exits after the first syntax error. It might be worthwhile, to study and investigate, whether this can be improved. It mainly depends on the possibilities available in javacc
- The current error reporting concerning errors discovered during type checking needs to be improved. Right now, only the line number in the source code is printed. In order to provide the proper fragment of the source code that caused the error, we need the end token in addition to the start token that is already recorded using the JavaParser method recordSourceCoordinates. Some suggestions are available through the JavaCC home page in the FAQ section.
6.4 What to pick first

Among the language constructs which are currently not available, type checking and code generation for declaring, throwing and catching exceptions are the most important to be added first. This will allow us to compile a much larger number of Java programs with Espresso.

The second most important deals with type checking and code generation for array initializers.

Both topics are suitable as part of a practical project in compiler design.

7.0 How to get involved in the project

The following section describes the procedure you have to follow, in order to get access to the Espresso source code, and what changes you have to make to your environment before fetching the sources from CVS. It also establishes a few basic rules you have to stick to, when updating CVS with modified versions of Espresso source files.

7.1 How to get access to Espresso’s most recent source code

Currently the Espresso source code is maintained on a local machine named *quasar* using CVS. This machine is Santiago M. Pericas (email: santiago@cs.bu.edu) private machine. He has volunteered to play the role of a “supervisor” for Espresso. Follow these steps to get involved:

- Send an email to Santiago M. Pericas asking for an account on node *quasar*
- Once you got the account, you need to set up the following environment variables to get ready for a CVS checkout
- Go to your home directory and type `cvs checkout espresso`
  This will checkout all sources, binaries and shell scripts into a directory tree whose root is `home/espresso`
- Add `$HOME/espresso/bin` to your path variable. This will make javacc and some useful shell scripts available to you, that you will need to either build (make) Espresso, and run it as your Java compiler.
- Next you need to construct Espresso’s lexer and parser using javacc. Use the command `javacc -STATIC=false java1.0.2.jj` in directory `espresso/parser`
- If you want to use CosmoCode as your Java Development Environment, use the CosmoCode project file `Espresso.proj` in directory `espresso` and edit manually the directory path for the Java source files to match to your local structure.
How to get involved in the project

- If you don’t want to use CosmoCode, you will need to compile Espresso by using a make file contained in directory espresso. Use command:
  
  `make -f Makefile`

- In either case, if the compilation was successful, Espresso is now available through the shell script `espresso`.

7.2 How to obtain a particular version of Espresso from CVS

If you want a particular version of Espresso for testing, proceed as above, but to check-out from CVS, use command: `cvs checkout -r versionid espresso`, where `versionid` is the name of the version you want to test. (Ask Santiago M. Pericas for available versions).

7.3 What are the rules for committing changes back to CVS

In order to commit changes back to CVS, you need to do the following:

- Make sure you tested your changes before
- If you made substantial changes, you might show your results first to Santiago M. Pericas
- Issue the command `cvs commit` from Espresso’s root directory `espresso`. Under all circumstances **add a comment stating the changes you made. When cvs is prompting for it.** This will allow us to identify, who changed what and for what reason, in case something gets messed up.

7.4 Espresso’s directory structure

This section introduces to the project structure of Espresso, as it is defined in CVS. The structure is described in terms of the directory structure, which results, when the entire project is checked out from CVS.

The main purpose of this section is, to help a user navigate through the different parts of the compilers source code.

Espresso was developed using Silicon Graphics Java Development Environment CosmoCode. The use of CosmoCode is optional, but highly recommended. In order to use CosmoCode, an appropriate project file defining the project structure needs to be defined. A template project file named `Espresso.proj` is managed within CVS and checked out to directory espresso. You will need to edit the file paths, in order to use it. Table 11 on page 41 shows Espresso’s directory structure together with the groups of files, which are located there.
How to get involved in the project

TABLE 11. Espresso's Directory structure

<table>
<thead>
<tr>
<th>Directory</th>
<th>Content/Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>espresso</td>
<td>Espresso.java</td>
</tr>
<tr>
<td>espresso/classfile</td>
<td>Constants.java</td>
</tr>
<tr>
<td>espresso/classfile/javaclass</td>
<td>Files provided by package available under GNU’s public license agreement, to load existing Java Class Files</td>
</tr>
<tr>
<td>espresso/classfile/classgen</td>
<td>Files provided by package available under GNU’s public license agreement, to generate Byte Code and to store to Java Class Files</td>
</tr>
<tr>
<td>espresso/classfile/util</td>
<td>Some utilities provided with classgen package</td>
</tr>
<tr>
<td>espresso/espresso</td>
<td>Root Directory of Espresso's Class Files</td>
</tr>
</tbody>
</table>
| espresso/parser                | java1.0.2.jj
(File containing definitions for Espresso's lexer and parser).

To generate JavaParser.java, JavaParserTokenManager.java, ... from it type javacc - STATIC=false java1.0.2.jj
JavaImportManager.java (JIM)

| espresso/syntaxtree            | Java sources for classes used to represent the AST                             |
| espresso/util                  | Java sources representing Espresso's type hierarchy                           |
|                                | Java sources used for Espresso's symbol table                                 |
|                                | Java sources used for error handling                                          |
|                                | Helper classes for scope stack and label stack                                 |
| espresso/bin                   | Contains the shell scripts you need for javacc and espresso                    |
| espresso/JavaCC                | Directory containing code for Java Compiler Compiler.                         |