Abstract

Compiler optimizations have been the target of research for many years. Java has received a lot of this attention due to its object-oriented nature and its many advanced features that lead to inefficient generated code. Many optimization frameworks, such as Soot, exist with the goal of optimizing Java through an intermediate representation. In this paper, we present X-JOAF which is an experimental framework similar to Soot, but deals with an untyped three-address code Intermediate Representation. This paper presents the core infrastructure of X-JOAF, and details of the transformations involved as well as the results from this experiment.

Categories and Subject Descriptors    D.3.2 [Java]; D.3.3 [Language Constructs and Features]: Data types and structures, Frameworks, Modules, Packages

General Terms    Design, Experimentation, Languages

Keywords    Java, Compiler Optimization, Intermediate Representation, Optimization Frameworks

1. Introduction

Compiler optimizations have been the target of compiler research over the past years. Optimization techniques that improve the efficiency of the execution of the produced machine code have been developed in the past. Most of this research has been geared towards optimizing Java bytecode. This can be attributed to many reasons, the most obvious being Java’s great popularity as an object-oriented programming language. Java is one of the most widespread programming languages, and has been increasingly used in the past years. However, the problem is that due to the advanced features of Java, such as execution safety and garbage collection, it tends to be slower than other programming languages [1]. Therefore, optimizing Java bytecode to exploit these important features without compromising its performance when compared to other languages is important. The other reason Java has received so much attention in the optimization research area is that optimizing bytecode can be a very tedious and difficult task to do due to its stack-based nature. Therefore, coming up with an appropriate Intermediate Representations (IR) and techniques which facilitate this job is important.

This technique of optimizing Java is a static optimization technique. Although static optimization techniques do not take Java’s dynamic nature into account, they do however sufficiently help in improving the performance of the produced code. Frameworks that provide such optimization functionality have been developed within many research groups, each using a different IR that is used to perform the optimizations. In this paper, we present X-JOAF which is a Java optimization analysis framework based on an untyped three-address code IR. X-JOAF is an experimental framework that aims to provide the core functionalities of an optimization analysis framework with the key feature of being modular and easily extensible in nature to allow it to grow easily in the future.

The rest of this paper is organized as follows. Section 2 discusses some related work that includes the different types of IR and some of the optimization frameworks currently available. Section 3 presents the motivation behind developing X-JOAF. Section 4 includes the implementation details of X-JOAF which describe the framework infrastructure. Section 5 presents the details of the transformations, and Section 6 gives an evaluation of X-JOAF. Section 7 describes the results of
our experiment while Section 8 presents a discussion of various issues. Finally, Section 8 and Section 9 present possible future work and the conclusion of this paper respectively.

2. Related Work

Since IRs are the common way to represent Java bytecode for optimization, a brief overview of the common types of IR is necessary here. There are various types of IR that are commonly used to optimize Java. These include stack-based representations, three-address code representations, and Single Static Assignment (SSA) representations.

The most obvious example of stack-based representations is Java’s own bytecode. Another stack-based representation is Baf which is part of the Soot framework [2]. Although Baf is stack-based, it may be considered a bit more readable than Java’s original bytecode. Stack-based IR are compact in their nature, and have the advantage of having implicit variable names. The problem with this kind of IR though is that it is not intuitively understandable due to the many computations that take place on the stack and is therefore difficult to perform optimizations upon.

The second type of IR mentioned above is the three-address code IR which is the type of IR used in X-JOAF. Three-address code IR has a maximum of three operands per operation i.e. one operand on the left hand side, and a maximum of two on the right-hand side. The advantage of three-address code representation is that it is easy to understand because of the limitation of the number of operands in any operation. Moreover, it is easier to perform optimizations upon compared to stack-based representations [3]. An example of such an IR would be Jimple that is used to perform optimizations in the Soot framework [2].

Finally, the SSA form which basically limits each variable to have one definition. It is a commonly used type of IR since it facilitates some optimizations such as constant propagation, global value numbering, partial redundancy elimination and register allocation due to its nature. This single definition nature of SSA helps in the calculations of the def-use chains of local variables and thus facilitates these mentioned optimizations. The problem with SSA, however, is the difficulty of calculating the \( \phi \) functions that determine the value of some local variables, and also the difficulty of converting out of SSA in order to be able to generate machine code afterwards. Shimple is an example of an SSA based IR. The Shimple framework was integrated into Soot at a later stage and includes Simple Shimple, Extended Shimple and Array Shimple [4].

There are other IRs that have been developed in the research community although the above mentioned three may be the most common ones. For example, [5] suggests an XML-based IR that can be used to optimize Java. Such representation facilitates the process of annotating the code with analysis results and runtime data while also being easily extensible due to its XML-based format. Other developed IRs are discussed in [6] which describes Pegasus, an IR used for imperative languages. Pegasus combines predicate SSA and gated SSA in its representation.

Since optimizing bytecode is not a very enjoyable task as previously mentioned, there have been several frameworks developed for optimizing it through an IR. Examples of these frameworks include the T.J Watson Libraries for Analysis (WALA) which was an IBM project initially. WALA supports both Java and JavaScript and uses an SSA-based IR [7]. It provides general analysis utilities and data structures that aid in optimizations of the original code. WALA also contains significant support for JEE semantics.

Scale is also another example of a framework developed for optimizations purposes [8]. Scale which is a scalable compiler for analytical purposes has been developed at the University of Texas at Austin and the University of Massachusetts Amherst. Scale uses a CFG-based IR, Scribble. However, optimizations and analyses are performed on an SSA-based form of Scribble. The framework provides easy reordering of existing transformations and the ability to create new transformations/optimizations.

Finally comes the Soot framework which is a very popular framework for optimizing Java bytecode using an IR. Soot was started by the Sable Research Group in McGill University in 1998, and has three different IRs that are used at different stages of the optimization process [2]. These include Baf, Jimple, and Grimp. Baf is a stack-based IR that eliminates some of the stack complications present in bytecode. Jimple is a three-address code representation that completely eliminates the stack-based, and is easily read to facilitate optimizations. The optimization analyses are usually done on Jimple. Finally, Grimp serves as a more readable IR format that also serves as the foundation for decomp-
lation purposes in the Soot framework [2]. Soot offers a stand-alone tool to optimize or inspect Java classfiles. It also provides a framework to develop optimizations or transformations on the Java bytecode. Other compiler optimization frameworks have also been implemented with different types of analyses in mind such as SPARK in [9] which was also developed by the Sable Research Group at McGill University. SPARK performs points-to analysis for Java code, and has also been integrated into Soot.

3. Motivation

Soot is a very large framework with a plethora of features that have been used over the years and that are still being added to currently. Despite that, there are still several motivations for developing a new analysis framework that is similar to Soot.

Soot has been developed over a ten year period in a research group with members coming and leaving every few years. This caused some parts of the framework to be properly documented while others were not which makes Soot very difficult to understand in order to develop any extensions for. Moreover, as more features were added to Soot over those years, the core infrastructure was never redesigned to be better suited for these new features. This made the framework become really huge and very difficult to understand. It is also very hard to keep track of the flow of execution in Soot, and to understand what each package or class is responsible. This is as a result of the different styles of naming in it which are not very intuitive to the purpose of the classes. Finally, Java itself, in which Soot is written, has evolved over the past ten years, and it would be very beneficial for Soot’s core to be redesigned to take advantage of the new features in Java and become more object-oriented.

Thus, the purpose of X-JOAF is to develop the core of an optimization framework that is more modular in nature than Soot, and which can be easily understandable and extensible with the vision that it may grow into a fully-functional framework similar to Soot. However, X-JOAF currently also serves as an experimental framework. This involves exploring the design decisions that may be imposed on the designers of such a framework if the IR grammar being used becomes untyped through untying the local variables.

4. X-JOAF

X-JOAF was designed with the goal of making it easily extensible for future use. The goal is to develop the core of an optimization framework that can be expanded to include more features that facilitate compiler analyses work. Currently, the core functionality of such a framework is implemented. This includes the structure of the grammar used for the IR, the internal data structures representing this grammar and the conversion from bytecode to the grammar and back to bytecode.

4.1 Grammar Description

X-JOAF uses a three-address code IR called EJimple (Extended Jimple). The grammar is based on Jimple from Soot [3] with just a few modifications that are intended for the experimental work done in this framework. Appendix A shows EJimple, the current grammar being used. Following are the modifications that have been made to the Jimple grammar to produce EJimple which are also summarized in Table 1. For the original Jimple grammar, please see [3].

4.1.1 Grammar Untyping

For experimental purposes, the local variable types from the IR grammar were removed making the grammar an untyped one. The types were left in method signatures to differentiate between overloaded functions. This mainly changed in the declaration rule in the grammar.

4.1.2 Method Arguments Limitation

To facilitate some analysis that may depend on the method parameters, we decided to limit method calls to only take locals as their parameters. This is because there will only be one case, a local variable, for any method parameter. This removes the need to handle multiple cases of a local variable or a constant for example, and it also makes the IR more readable.

4.1.3 Expressions Hierarchy

In Jimple, there was no differentiation between arithmetic expressions and boolean expressions while this distinction exists in Java. Therefore, we thought that it would be semantically clearer to divide binary expressions into arithmetic expressions and boolean expressions. This will restrict boolean conditions to only take a boolean expression and thus make the IR more readable and closer to the mindset of a Java programmer.
Table 1. Changes to Jimple Grammar

<table>
<thead>
<tr>
<th>Jimple</th>
<th>EJimple</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaration = jimple_type local_name_list semicolon;</td>
<td>declaration = var local_name_list semicolon;</td>
</tr>
<tr>
<td>arg_list = immediate</td>
<td>immediate comma arg_list;</td>
</tr>
<tr>
<td>binop_expr = immediate binop immediate;</td>
<td>binop_expr = bool_expr</td>
</tr>
<tr>
<td></td>
<td>bool_expr = value bool_binop value;</td>
</tr>
<tr>
<td></td>
<td>arithmetic_expr = value arithmetic_binop value;</td>
</tr>
</tbody>
</table>

4.2 Framework Infrastructure

In order to implement the core of the framework, there were two different sets of data structures required. The first was a set to encapsulate the data in the Java binary class files into Java objects that can be dealt with easily within the framework. This includes a hierarchy for the bytecode instructions and some data structures to represent other information present in the classfile. This was done using the Byte Code Engineering Library (BCEL) [10]. The second set is the data structures that will represent all the production rules in the EJimple grammar so that the bytecode can be easily converted into internal EJimple objects that can be used for optimizations and for conversion back to bytecode in the last stage of the framework.

4.2.1 BCEL

The first stage in a framework that takes a Java classfile as its input is to read this classfile into a format that is easily understandable and easy to use internally in Java. Thus, representing each classfile attribute as a Java class, and designing a hierarchy for the different bytecode instructions was a natural line of thought. However, this is not a new idea and this encapsulation is a process that has been implemented repeatedly over the past years. As this part of the framework is mainly a utility we need to facilitate the functionality of the rest of the framework, there was no need to re-invent the wheel. Thus, an existing library, BCEL, was used. BCEL is a library developed by Apache [10] which allows its user to analyze and manipulate Java class files. The library basically parses the class file and produces an encapsulation of all its information and methods in a JavaClass object which can then be used for our purposes. The library also provides the feature of generating binary files from scratch using the ClassGen class which is a very useful feature when converting from our IR back to bytecode.

4.2.2 EJimple Representations

In order to allow optimizations to be done to the IR, and in order to make the framework modular for any functionalities that may be added to it, the IR grammar structure has to be represented internally in Java. Thus, all production rules are represented by Statement or Expression classes. The details of the various packages and their functionalities are explained below.

Constants Package This package contains all the types of constants that are supported in the Java language. Although BCEL provides a constants package for the constants found in the classfile, we could not use that package as is because all their Constant classes were based on their reference in the constant pool which was irrelevant once the IR representation stage was reached. Thus, we used our Constants class hierarchy, and converted BCEL constants to our Constant classes. The super class Constant inherits from Value so that a constant can appear in any expression.

Expressions Package This package contains all the expression productions that are in the EJimple grammar. The superclass Expression inherits from Value since the right-hand side of a DefinitionStatement, the most commonly used statement, is a Value. This value can be a Constant, and Expression or a Local. Thus, the Local class also inherits from Value. Having this inheritance hierarchy also provides a flexibility in the level of abstraction needed to deal with on the operand stack that is used during the conversion from bytecode to EJimple.

Statements Package This package represents the statements in the grammar. Statements are basically the content of any EJimple method. Bytecode instructions get translated into their corresponding statements to form the body of a method.
Transformations Package  This package contains the class responsible for the transformations from bytecode to EJimple and back. The EJimpleTransformer is the starting engine of the first transformation which takes a classfile as its input. This produces an EJimpleFile object which represents the transformed classfile in its IR form. Any optimizations should be done on this EJimpleFile object before it is fed again to the BytecodeTransformer to be transformed back to bytecode. Currently, we do not support any optimizations yet, but understanding the flow of execution in the framework should help the future work in this direction. Figure 1 shows this flow. It is also to be noted that jsr and ret bytecode instructions are not supported in our framework, and thus have no corresponding states for them. This is because dealing with these two instructions is very difficult and will cause the generated code to become messy. Moreover, these instructions are no longer generated in the bytecode as of Java 1.4.

5. Transformations
5.1 Bytecode to EJimple
Transforming bytecode to EJimple is done in a systematic fashion where each bytecode instruction in a method is transformed into a Statement in the corresponding EJimpleMethod. However, there are some calculations that need to be done first before the direct transformation from BCEL instructions to EJimple statements. The general idea is that the bytecode instructions are going to be parsed one by one, and an operand stack will be used to simulate the stack operations that occur in the bytecode. The three steps needed to do the transformation are described below, and the reasons behind needing the first two steps are explained. Figure 5.1.3 shows the whole algorithm needed for the transformation.

5.1.1 Generating the Local Variables Map
The first step is determining the local variables that belong to the current method being translated as these will be used in any store or load bytecode instructions. Each method in the bytecode has a local variable table that holds the local variables in this method. All store and load instructions use an index into this table to specify what local variable is being referenced by this instruction. Each local variable specified in this table has a Local object created for it in our implementation. Since EJimple is an untyped grammar, we do not know the type of the Local objects created and thus cannot base their index on their size as what happens in the bytecode. However, if the indices of the created EJimple locals are not in accordance with what is in the bytecode, the translation will not be accurate. Thus, we need to map the index of the local variable as found in the bytecode to the Local object generated to represent it so that we can easily find the local object referenced during the transformation. This is done by generating a hashmap from an Integer which represents the local variable’s index to a Local object. At the beginning of each method, this hashmap is generated based on the local variables available in the method as indicated from BCEL. Lines 2 to 4 in the algorithm below show this step.

5.1.2 Generating the Label Map
The next problem that needs to be taken care of is keeping track of the labels. All branch instructions in the bytecode specify the position/address of the instruction they are going to branch to whether it is a conditional or unconditional branch. If we consider the example in Figure 2 and assume the normal instruction by instruction processing then when the goto instruction at position 2 is reached, we will generate a label for this instruction and move to the next one. When we reach the instruction at position 8 which was the target of that goto instruction, we will have no indication that a label for this instruction is needed. Assuming we can keep track of the generated labels and the positions they are generated for, then the problem of this goto will be
solved. However, if we reach the conditional branch instruction at position 10, this will not work as we only proceed forward during the translation, and will thus never go back to position 5 to insert the generated label there. Moreover, even if we had a technique to go back, we will not be able to determine if this label is generated for an instruction that will still appear or one that has already appeared i.e representing a back edge.

Thus, we need to have all the labels for the method determined before we begin the actual translation process. This has been done by generating a hashmap that maps the position of an instruction to the label generated for it. This mapping process is done through an initial a pass over the bytecode instructions in the method before actually parsing them. Since try and catch blocks are also translated as branch instructions in the bytecode, a label is generated for the beginning of each try block and the beginning of each handler block. This means that the exception table in each method should also be considered. Lines 5 to 13 in the algorithm below [5.1.3] show this process. After this map is generated, the normal instruction by instruction translation can be done. Details on how this generated map is used during the translation are explained in the next step.

5.1.3 Producing the Statements

After the previous two steps are performed, the instruction list for the current method is looped upon. Before determining the type of the current instruction to generate the corresponding statement for it, the position of this current instruction is looked up in the map generated in step 2. If an entry exists, then a LabelStatement with the label found for this entry is generated before the actual statement corresponding to this instruction is generated. Then, the type of instruction is determined, and the corresponding type of statement is generated for it. Lines 14 to 17 in the algorithm below [5.1.3] show this. Since the EJimple grammar is untyped while the bytecode instructions are typed, we need to have the type of the current instruction stored somehow so that it can be used when translating from EJimple back to bytecode. Thus, we made our statements and expressions typed so that when the statements are parsed for the second set of translations, the right bytecode instruction can be generated for them.

During parsing the bytecode instructions and generating corresponding EJimple statements, the operations that are normally done by the operand stack in the bytecode are done on an operand stack of Local objects used for our purposes. For example, when a load instruction is encountered, a definition statement is generated for it which assigns the value of the local variable being loaded to a generated temporary variable. Since what happens in the bytecode is that the value of the local variable is loaded on the stack, then in our case, the temporary variable generated in this definition statement will be pushed onto the stack. The opposite happens for a store instruction for example.

However, we faced a problem with the DUP instructions that caused us to change the structure of the operand stack we are using for the conversion. The problem is that the functionality of certain DUP instructions depends on the type of the variable on the stack. Consider the DUP2 instruction for example, and assume word1 and word2 exist on the top of the stack. DUP2 would duplicate both words and insert the copies on top of the stack. However, word1 and word2 may represent two separate values or may represent one value if it is a Long or Double value. This is a problem in our case since our stack will represent both these cases as one local variable independent of its size or type. Since our local variables are not typed, then we cannot know the size to determine how many local variables will need to be duplicated on our stack. Thus, we need to know the size of the local variables on our operand stack.

Unfortunately, this is not the only problem with the DUP instructions. Given the way that the translation from bytecode to EJimple is done, then a DUP instruction will be translated into a definition statement which looks like this for example: $temp4 = $temp3 When the second round of transformation comes in to transform this statement into bytecode, this will simply be transformed into a load instruction if no optimizations are done to generate DUP4 instructions directly. In order to determine the type of the load instruction, the definition statement must have a type like all our definition statements do, but the problem is we do not the type of the original local variable on the stack which we are duplicating, and therefore will not be able to determine a type for this statement. Thus, not only do we need to know the size of the local variables on the stack, we also need to know their type.

Although the obvious solution would be to simply type the local variables themselves, we wanted to con-
continue with our experimentation of an untyped grammar and therefore did not yield to this solution. Since only keeping track of the sizes of the local variables on the stack will not solve the second problem discussed, we decided to keep track of the type of the local variables on the stack. This was done by creating a class called `OperandStackEntry` which contains a `Local` object and a `Type` object that specifies the type of this local variable. Objects of this new class are those pushed on the stack. Thus, we now keep track of the types of the local variables that are pushed on the stack.

6. Results

Our experiment shows that having an untyped grammar representing Java bytecode was not a successful design decision because Java bytecode in itself is typed. Therefore, when converting EJimple back into Java bytecode, we need to know the type of the variable we are dealing with in order to produce the right bytecode instruction. In our experiment, we ended up typing statements and expressions to implicitly type the local variables.

Since we obtain the type of the statements from the original bytecode, the current implementation with the current grammar specifications cannot support someone writing an EJimple file from scratch and converting it into bytecode. This is because the grammar does not currently provide any means to implicitly type statements as what happens in the internal representation of EJimple. A discussion of some suggestions here will be mentioned in the next section.

Finally, and most importantly given the nature of this framework as an optimization one, the currently produced EJimple code is inefficient. When compared to Jimple, the size of our produced IR will be much more due to the fact that jimple has many optimizations performed in it that are directly implemented in the Soot framework itself. Currently, however, there are no optimizations done on the produced EJimple code yet, and the transformation process takes place by simulating the bytecode operand stack which causes many temporary variables to be generated. Possible solutions to change this will be provided in the discussion section.

7. Discussion

There are several techniques that could have improved the efficiency of the produced EJimple code that were not implemented in our experiment. The first one would have been to implement an implicit constant propagation optimization analysis during the transformation.
This could have been done by directly pushing the actual constant onto the operand stack instead of generating a temporary variable for it and then pushing that temporary variable onto the stack. This would have greatly reduced the number of temporary variables generated. However, we could not implement that due to the restriction of having method call arguments only accept local variables. If constants were allowed to exist on the stack, then a constant may be used as an argument to a method call which would violate this restriction. Since we could not perform the constant flow propagation during the transformation, performing it on the produced IR could greatly help. Unfortunately, due to time constraints, this was not implemented. However, if constant flow propagation analysis as well as copy flow propagation optimization are performed on the produced IR, this will greatly improve it in terms of its size and efficiency in the bytecode that would be produced from it.

Finally, in EJimple, we untyped the grammar by untyping the local variables. This made us end up typing the statements and expressions in the internal Java representation of the grammar. Other similar interesting experiments could include untyping the local variables, but providing typed operators in the grammar itself. For example, adding two integers would involve the operator $iadd$ or $iplus$. This would allow us to know the types of the variables in that operation and would allow someone to write an IR file from scratch and transform it into bytecode since the types are now explicitly put into the expressions and statements through the typed operators. More investigation in this direction might yield more encouraging results that what was produced in this experiment.

8. Future Work

Currently, only the core infrastructure of X-JOAF is implemented along with the transformation from bytecode to EJimple and back to bytecode. The optimization infrastructure itself still needs to be implemented in order to be able to optimize the produced EJimple, and to allow new optimization analyses to be implemented in the framework. Additionally, the grammar could become typed once again and be used for X-JOAF in order to make the framework useful, and encourage other researchers to expand on it in order to reach the functionalities of a framework similar to Soot. Finally, we need to modify our framework to accept EJimple as input source file. This might lead to some modifications in the grammar file of EJimple.

9. Conclusion

In this paper, we presented X-JOAF, an extensible Java optimization framework. This framework was an experimental one that explored the design implications that may be imposed on the implementer of such a framework if an untyped IR grammar was used. The results of this experiment showed that since the Java bytecode in itself is typed, using a completely untyped grammar is not possible. In our case, we ended up implicitly typing the grammar through the statements and expressions. Thus, the types have to be kept track of somehow in order to be able to convert from the IR back to bytecode. However, the basic infrastructure for such an optimization framework is implemented in X-JOAF, and changing it to support a typed grammar once more should not be a difficult task.

Acknowledgments

We would like to thank Ondřej Lhoták for suggesting this experiment to us which enabled us to explore this kind of work in more depth and understand the different angles to it. We would also like to thank him for his guidance throughout the project.

References


A. EJimple Grammar File

Package ca.uwaterloo.xjoaf.ejimple.parser;

Helpers

all = [0 .. 0xffff];

dec_digit = [‘0’ .. ‘9’];
dec_nonzero = [‘1’ .. ‘9’];
dec_constant = dec_digit+;

hex_digit = dec_digit | [‘a’ .. ‘f’] | [‘A’ .. ‘F’];
hex_constant = ‘0’ (‘x’ | ‘X’) hex_digit+;

oct_digit = [‘0’ .. ‘7’];
oct_constant = ‘0’ oct_digit+;

quote = ‘”’;
escapable_char = ‘\‘ | ‘' | quote | ‘.’ | ‘#’ | ‘”’ | ‘n’ | ‘t’ | ‘r’ | ‘b’ | ‘f’;
escape_code = ‘u’ hex_digit hex_digit hex_digit hex_digit;
escape_char = ‘\‘ (escapable_char | escape_code);

not_cr_lf = [all - [10 + 13]]; 
not_star = [all - ‘*’];
not_star_slash = [not_star - '/'];

alpha_char = [‘a’ .. ‘z’] | [‘A’ .. ‘Z’];

simple_id_char = alpha_char | dec_digit | ‘_’ | '$';

first_id_char = alpha_char | ‘_’ | '$';

quotable_char = [not_cr_lf - ‘”’];

string_char = escape_char | [0 .. 33] | [35 .. 91] | [93 ..127];

line_comment = ‘‘ not_cr_lf*;
long_comment = ’/*’ not_star* ’+*’ + (not_star_slash not_star* ’+*’+ ’*/’;

blank = (‘ ’ | 9 | 13 | 10)+;
ignored_helper = (blank | line_comment | long_comment)+;

Tokens

ignored = ignored_helper;
abstract = ‘abstract’;
final = ‘final’;
native = ‘native’;
public = 'public';
protected = 'protected';
private = 'private';
static = 'static';
synchronized = 'synchronized';
transient = 'transient';
volatile = 'volatile';
strictfp = 'strictfp';
enum = 'enum';
annotation = 'annotation';

class = 'class';
interface = 'interface';

var = 'var';

void = 'void';
boolean = 'boolean';
byte = 'byte';
short = 'short';
char = 'char';
int = 'int';
long = 'long';
float = 'float';
double = 'double';
null_type = 'null_type';

extends = 'extends';
implements = 'implements';

breakpoint = 'breakpoint';
case = 'case';
catch = 'catch';
cmp = 'cmp';
cmpg = 'cmpg';
cmpl = 'cmpl';
default = 'default';
entermonitor = 'entermonitor';
exitmonitor = 'exitmonitor';
goto = 'goto';
if = 'if';
instanceof = 'instanceof';
interfaceinvoke = 'interfaceinvoke';
lengthof = 'lengthof';
lookupswitch = 'lookupswitch';
neg = 'neg';
new = 'new';
newarray = 'newarray';
newmultiarray = 'newmultiarray';
nop = 'nop';
ret = 'ret';
return = 'return';
specialinvoke = 'specialinvoke';
staticinvoke = 'staticinvoke';
tableswitch = 'tableswitch';
throw = 'throw';
throws = 'throws';
virtualinvoke = 'virtualinvoke';
null = 'null';
from = 'from';
to = 'to';
with = 'with';
cls = 'cls';

comma = ';';
l_brace = '{';
r_brace = '}';
semicolon = '; ';
l_bracket = '[';
r_bracket = ']';
l paren = '(';
r paren = ')';
colon = ':';
dot = '.';
quote = '"';
colon_equals = ':=';
equals = '=';
and = '&';
or = '|';
xor = '^';
mod = '%';
cmpeq = '===';
cmpne = '!==';
cmpgt = '>'; 
cmpge = '>=';
cmplt = '<'; 
cmple = '<=';
shl = '<<';
shr = '>>';
ushr = '>>>';
plus = '+';
minus = '-';
mult = '*';
div = '/';
full_identifier = ((first_id_char | escape_char) (simple_id_char | escape_char)* \'.\')+
  (first_id_char | escape_char) (simple_id_char | escape_char)*;
quoted_name = quote quotable_char+ quote;
identifier = (first_id_char | escape_char) (simple_id_char | escape_char)* | 
  \'<clinit>' | \'<init>'

at_identifier = '@' (\'parameter\' dec_digit+ \'\:' | \'this\' \':\' | \'caughtexception\');

bool_constant = 'true' | 'false';
integer_constant = (dec_constant | hex_constant | oct_constant) \'L\'?;
float_constant = ((dec_constant \'.\' dec_constant) ((\'e\' | \'E\') (+\'\:' | \'-\')? dec_constant)? (\'f\' | \'F\')?) |
  (\'#\' ((\'\:-\' ? 'Infinity') | 'NaN') (\'f\' | \'F\')? )
string_constant = """ string_char* """;

Ignored Tokens
  ignored;

Productions
  file = modifier* file_type class_name extends_clause? implements_clause? file_body;

    modifier =
      \{abstract\} abstract |
      \{final\} final |
      \{native\} native |
      \{public\} public |
      \{protected\} protected |
      \{private\} private |
      \{static\} static |
      \{synchronized\} synchronized |
      \{transient\} transient |
      \{volatile\} volatile |
      \{strictfp\} strictfp |
      \{enum\} enum |
      \{annotation\} annotation;

    file_type =
      \{class\} \[theclass\]:class |
      \{interface\} interface;

    extends_clause = extends class_name;

    implements_clause = implements class_name_list;

    file_body = l_brace member* r_brace

    name_list =
      \{single\} name |
      \{multi\} name comma name_list;
class_name_list =
    {class_name_single} class_name |
    {class_name_multi} class_name comma class_name_list;

member =
    {field} modifier* type name semicolon |
    {method} modifier* type name l_paren parameter_list? r_paren throws_clause? method_body;

type =
    {void} void |
    {novoid} nonvoid_type;

parameter_list =
    {single} parameter |
    {multi} parameter comma parameter_list;

parameter = nonvoid_type;

throws_clause = throws class_name_list;

base_type_no_name =
    {boolean} boolean |
    {byte} byte |
    {char} char |
    {short} short |
    {int} int |
    {long} long |
    {float} float |
    {double} double |
    {null} null_type;

base_type =
    {boolean} boolean |
    {byte} byte |
    {char} char |
    {short} short |
    {int} int |
    {long} long |
    {float} float |
    {double} double |
    {null} null_type |
    {class_name} class_name;

nonvoid_type =
    {base} base_type_no_name array_brackets* |
    {quoted} quoted_name array_brackets* |
    {ident} identifier array_brackets* |
    {full_ident} full_identifier array_brackets*;
array_brackets = l_bracket r_bracket;

method_body =
    {empty} semicolon |
    {full} l_brace declaration* statement* catch_clause* r_brace;

declaration = var local_name_list semicolon;

local_name = name;

local_name_list =
    {single} local_name |
    {multi} local_name comma local_name_list;

statement =
    {label} label_name colon |
    {breakpoint} breakpoint semicolon |
    {entermonitor} entermonitor value semicolon |
    {exitmonitor} exitmonitor value semicolon |
    {tableswitch} tableswitch l_paren value r_paren l_brace case_stmt+ r_brace semicolon |
    {lookupswitch} lookupswitch l_paren value r_paren l_brace case_stmt+ r_brace semicolon |
    {assign} variable equals expression semicolon |
    {if} if bool_expr goto_stmt |
    {goto} goto_stmt |
    {nop} nop semicolon |
    {return} return value? semicolon |
    {throw} throw value semicolon |
    {invoke} invoke_expr semicolon;

label_name = identifier;

case_stmt = case_label colon goto_stmt;

case_label =
    {constant} case minus? integer_constant |
    {default} default;

goto_stmt = goto label_name semicolon;

catch_clause = catch [name]:class_name from [from_label]:label_name to [to_label]:label_name
with [with_label]:label_name semicolon;
expression =
  {new} new_expr |
  {cast} l_paren nonvoid_type r_paren value |
  {instanceof} value instanceof nonvoid_type |
  {invoke} invoke_expr |
  {reference} reference |
  {binop} binop_expr |
  {unop} unop_expr |
  {value} value;

new_expr =
  {simple} new base_type |
  {array} newarray l_paren nonvoid_type r_paren fixed_array_descriptor |
  {multi} newmultiarray l_paren base_type r_paren array_descriptor+;

array_descriptor = l_bracket value? r_bracket;

variable =
  {reference} reference |
  {local} local_name;

bool_expr = [left]:value bool_binop [right]:value;

arithmetic_expr = [left]:value arithmetic_binop [right]:value;

invoke_expr =
  {nonstatic} nonstatic_invoke local_name dot method_signature l_paren arg_list? r_paren |
  {static} staticinvoke method_signature l_paren arg_list? r_paren;

binop_expr =
  {boolean} bool_expr |
  {arithmetic} arithmetic_expr;

unop_expr = unop value;

nonstatic_invoke =
  {special} specialinvoke |
  {virtual} virtualinvoke |
  {interface} interfaceinvoke;

method_signature =
  cmplt [class_name]:class_name [first]:colon [return_type]:type [method_name]:name l_paren parameter_list? r_paren cmpgt;

reference =
  {array} array_ref |
  {field} field_ref;

array_ref = identifier fixed_array_descriptor;
field_ref =
   \{local\} local_name dot field_signature |
   \{sig\} field_signature;

field_signature = cmplt [class_name]:class_name [first]:colon [field_name]:name cmpgt;

fixed_array_descriptor = l_bracket value r_bracket;

arg_list =
   \{single\} local_name |
   \{multi\} local_name comma arg_list;

value =
   \{local\} local_name |
   \{constant\} constant;

constant =
   \{integer\} minus? integer_constant |
   \{float\} minus? float_constant |
   \{string\} string_constant |
   \{boolean\} bool_constant |
   \{clzz\} [id]:class string_constant |
   \{null\} null;

arithmetic_binop =
   \{and\} and |
   \{or\} or |
   \{xor\} xor |
   \{mod\} mod |
   \{cmp\} cmp |
   \{cmpg\} cmpg |
   \{cmpl\} cmpl |
   \{shl\} shl |
   \{shr\} shr |
   \{ushr\} ushr |
   \{plus\} plus |
   \{minus\} minus |
   \{mult\} mult |
   \{div\} div;

bool_binop =
   \{cmpeq\} cmpeq |
   \{cmpne\} cmpne |
   \{cmpgt\} cmpgt |
   \{cmpge\} cmpge |
   \{cmplt\} cmplt |
   \{cmple\} cmple;
unop =
   \{lengthof\} lengthof |
   \{neg\} neg;

class_name =
   \{quoted\} quoted_name |
   \{ident\} identifier |
   \{full_ident\} full_identifier;

name =
   \{quoted\} quoted_name |
   \{ident\} identifier;