CS444/644 Assignment 1 Technical Report

Ho-Yi Fung, Jianchu Li, Zhiyuan Lin

April 4, 2016

Abstract

In the final part of the project we implement the code generation phase of the compiler. The implement has been completed and successfully passed all the Marmoset test cases. In this report we discuss and justify our design decisions when creating the code generator.

Chapter 1

Design

In this chapter we discuss the design of the code generation phase, and the problems we solved with our design. Section 1.1 discusses the conventions and implementation schemes we used in the code generation process. This includes the calling conventions, the stack frame (local variable storage), the class and object layout, and the signature scheme for generating labels. Section 1.2 discusses the major problems the team solved when designing the code generator.

1.1 Conventions

1.1.1 Calling Convention

A consistent subroutine calling convention is used in all the NASM code generated. The scheme implemented by the team requires that:

1. All arguments for a subroutine are stored on the stack.
2. The arguments are pushed onto stack from left to right (consistent with the order they are evaluated).
3. In the case where the subroutine is the implementation of an instance method, the object address needs to be pushed to the stack *after* all the arguments. In the case of a static method or a native subroutine, the value *0* (null) is pushed to the stack in place of object address to ensure that the arguments have consistent offsets. This value 0 should never be accessed.
4. The *caller* of a subroutine is responsible for saving registers and restoring registers when calling a subroutine.
5. The *callee* allocates its own stack frame for local variables and cleans the frame at the end of the subroutine.

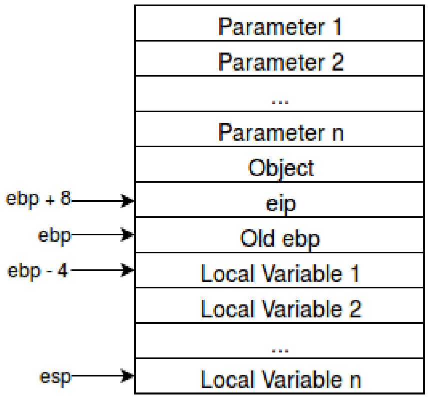

Figure 1.1: Stack Frame

6. The frame pointer of the caller is pushed on the stack by callee at the very beginning of the subroutine and restored before the subroutine exits.

7. The caller is responsible for cleaning up the arguments (including object address for instance method).

8. The return value of a subroutine is stored in the register *eax*.

We choose to push arguments onto the stack from left to right because this makes left-to-right evaluation of arguments straight-forward. However the object address of an instance method must be evaluated before any arguments to produce side effects consistent with the Java semantics. Therefore the object address is first evaluted and pushed on the stack, and then the arguments are evaluated. However, whenever an argument needs to be pushed onto stack, the code generated would pop the object address from stack, push the argument, and then push the object address back on the stack. This way the object is always kept on the top of the stack and has the same offset for all instance methods.

The order of arguments in this scheme is different from that in **cdecl** or **thisdecl** (used by C++ for instance methods), but because there is no variadic function (variable length argument) in Joos 1W, this scheme works correctly and is time efficient.

In general, this scheme applies no matter the subroutine generated is in effect a Java method or a helper subroutine used in the NASM code generated. The only exceptions are the subroutines provided by the library *runtime.s*, where we will store the input for the subroutine in register *eax*. This includes when the native method *PrintStream.nativeWrite()* is called.

1.1.2 Stack Frame and Local Variable Storage

As specified by the calling convention described in the last section, in a subroutine call the callee allocates and cleans the stack frame. Local variables are assigned addresses in reference to the frame pointer (*ebp*, also called base pointer in NASM). For example the first local variable declared in a Java method is stored in the address (ebp - 4), and the second is in (ebp - 8). The real offset starts from 4 because [ebp] is the old frame pointer from the caller stored by callee, following our calling scheme. Figure 1.1 shows the structure of the stack frame in our design.

Our scheme for computing variable offset is shown in Figure 1.2. The space for a variable is reused when it is out of scope. All parameters have negative offset which corresponds to the order that they are pushed on the stack. For example, because in our call convention the first argument *a* is pushed onto stack first, it has the lowest offset.

1.1.3 Class Layout

For the sake of clarity, we decide to generate an assembly file for each Java class given as input. Each assembly file for Java classe normally has the following components:

1. pointer to corresponding column in subtype table

2. pointer to corresponding column in interface table

3. Virtual method table for the class

```
public void m(int a, int b, int c) {
  // offset(a) = -3, offset(b) = -2, offset(c) = -1
  int i; // offset 0
  {
    int j; // offset 1
    int l; // offset 2
  }
  int k; // offset 1
}
```

Figure 1.2: Example of Local and Formal Offsets

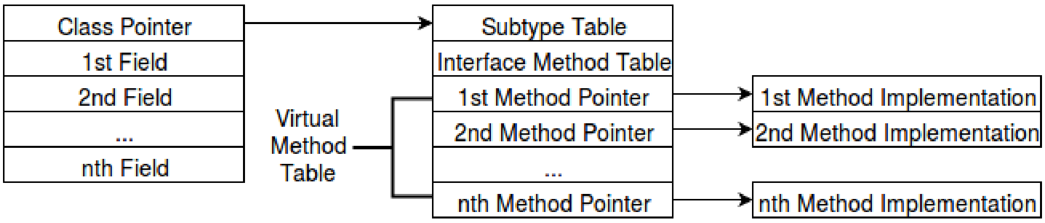

Figure 1.3: Object Layout

4. Class Fields

5. Separate initialization subroutines for instance and class fields

6. Implementation of methods

In the case where a *test()* method is specified in the class, a *_start* subroutine is also generated in the class assembly file. The *_start* subroutine initializes the class fields by calling the initialization subroutines for each class. Within each class the class fields are initialized in the order they are declared. After the initialization, the subroutine that implements the *test()* method in assembly is called. Each non-native method is implemented as a subroutine in assembly and follows the calling convention.

The virtual method table is used in order to support method overriding. It also contains pointers to tables for computing interface method offset and class hierarchy. The detail of the virtual method table will be discussed later in Section 1.2.3.

1.1.4 Object and Array Layout

Space for objects and arrays is allocated using the *__malloc* subroutine provided in the *runtime.s* library. Each object contains a pointer to its runtime class which contains the virtual method table. This pointer is stored as the first word in the object memory. In each object, the virtual table pointer is followed by the instance fields for the object.

Offsets for instance fields are calculated prior to the actual code generation process, and follows the inheritance relationship between classes to support instance field polymorphism. For example, if class A inherits the instance field x from class B, the field x has the same offset in both A and B objects.

Moreover, hidden fields need to be take into consideration when generating field offsets. In Java, both instance and static fields are inherited, but there is no field polymorphism. In other words, fields are hidden but not overriden. For example in Figure 1.4, the return value of *B.test()* should be 123. In this example the object a must contain space for the field x from both class A and class B. In order to support this feature, each class inherits the layout of its superclass and extends it with the fields declared in itself. Different fields with same name but are declared in different classes need to have different offsets.

Arrays have a similar layout with Java objects. The first word in an array points to a virtual method table, which then points to the table column for class hierarchy check. The second entry in an array stores the length of the array, and then the array elements are stored in order.

```
public class A {
  public int x = 123;
  public A() {}
}
```

(a) Class A

```
public class B extends A {
  public int x = 2;
  public B() {}
  public static int test() {
    A a = new B();
    return a.x;
    // should be 123
  }
}
```

(b) Class B

Figure 1.4: Example of Field Hiding

1.1.5 Label Signature Scheme

In the process of code generation we need to generate assembly labels for elements such as class fields, methods as well as virtual tables. A signature scheme was devised in order to generate globally unique labels. See below for some examples.

- (Static) Field: *typename#fieldname*
- Method: *typename#methodname$paramter_type1?parameter_type2?...$*
- Constructor: *typename#*⟨init⟩*$parameter_type1?...$*
- Primitive Type: I for *int*, C for *char*, B for *boolean*, V for *void*
- Class or Interface Type: fully qualified name
- Array Type: @*element_type*

The *typename* above is the fully qualified name of the type in which the member (field or method for example) is declared. Parameter type signatures follow the above scheme for types. For example the method *x(int[] a)* from class *A.B* has signature *A.B#x$@I$*.

This scheme is designed following the signature scheme used in JVM bytecode. However some delimiters are replaced because they cannot be used in names in NASM.

1.2 Code Generation

In this sections we discuss the design issues in actual code generation. Only challenges and points of interest are introduced here. The code generation for certain statements and expressions merely follows templates available from class and the internet. Such parts are not covered for the sake of brevity.

1.2.1 The Code Generation Process

We decide to follow the *Visitor* pattern when implementing the code generator, as we did for type checker and other previous modules. The code is generated from bottom up. For example when generating code for an expression, we would first visit its subexpressions in the order specified by JLS, and then incorporate the code snippets of subexpressions into the code for the expression.

The code generation process of many statements and expressions utilizes label for storing data or implementing control flow. Most of these labels are not global values, and only need to be unique within an assembly file. In these cases counters are used to produce unique labels.

1.2.2 Field Initialization and Access

As was discussed in Section 1.1.3 and Section 1.1.4, instance fields are stored in its object memory and static fields are stored under globally unique labels. Consequently the access of an instance field utilises the offset that has been computed for the field before code generation, while the access of a class field is achieved by means of its label.

All static fields are initialized in the *_start* subroutine before the implementation of *test()* is called. Instance fields are initialized before each constructor calls. Fields are initialized in the order that they are declared in the class, and any field that has not been initialized has the default value *0* (false and null are also represented as *0*).

1.2.3 Method Invocation

Accessing an instance method requires a slightly different approach because of method polymorphism. To support method overriding, class specific *virtual method table*s (vtable for short) are used. In our design, the virtual table of a class is prepended by two entries. The first entry of contains a pointer to the subtype table, which will be discussed later in Section 1.2.6, and the second entry contains a pointer to interface method table for interface method calls. Following these two entries are the real virtual method table, which contains the instance method labels in order consistent with their precomuted offsets. As was mentioned in Section 1.1.4, each object contains a pointer to the virtual table of its runtime class.

Method access has three main scenarios. The first is instance method call with an object whose static type is a class type. The second is when the object is of interface type statically. The third is static method access.

In the first case, similar to instance fields, offsets are computed for instance methods so that they could be accessed from the class' virtual method table. However because of polymorphism, if an instance method $A.x(...)$ overrides some method $B.x(...)$ from the super class, it must have the same offset as the replaced method, and thus takes its place in the virtual table.

In the case where the method is declared in an interface, the method cannot be accessed simply using the virtual method table, because there can be multiple interfaces, and coordinating the virtual table offsets of methods in multiple interfaces is inefficient. The interface method table is therefore used to facilitate method access in this case. The table is created in the form of selector index arrays. In the table there is a column for each class, and the pointer in each class points to their own column. Each method appearing in one or more interfaces is given a global offset, which indicates their positions in the class columns. An entry in the table is a pointer to the method implementation if the method is implemented by the class, otherwise it is *null* and should never be accessed. Given a interface instance method call, we find the column that belongs to the object's class, and then find the pointer to the method implementation through offset of the interface method. The interface method table is generated in its own assembly file.

Lastly, static method invocation is achieved in assembly by calling the globally unique label of the method implementation, and there is no method overloading for static methods in Java.

1.2.4 Constructor Calls

Constructors are invoked through their labels, just like static methods. Constructor calls also follow the calling convention for methods. However, space for the object must be allocated first through the *__malloc* subroutine and the returned pointer to object must be placed on top of all arguments for the constructor. Moreover, the subroutine for instance field initialization must be called before the subroutine that implements the constructor. Besides, each constructor implicitly calls the default constructor of its super class before running its own code.

1.2.5 Side Effects

The team takes care to evaluate expressions in the right order so that they have the expected side effects in Java. For example, in the case of a method call *o.m(a)*, as was mentioned in Section 1.1.1, the object *o* is evaluated first, and then the argument expression *a*, eventually the actual method call happens. Similarly for array access *a[i]*, the array *a* is evaluated first, and then the index expression *i*.

1.2.6 Dynamic Type Checking

Java requires type checking at runtime for the *cast* expression, *instanceof* expression, and *array access* expression. This is supported by the subtype table. The subtype table is similar to the interface method table as it contains a column (array) for each type. However in this case columns for array types are included too. When checking if an object a is an instance type T, we access the subtype table through a's pointer to its class, and then the pointer in its class to the subtype table column. Each type is also associated with an offset in the subtype table. Therefore we could retrieve T's offset in compile-time and access its entry in a's column based on the offset. The entry is *true* if a is an instance of T, and *false* otherwise. The checking for the cast expression goes through the same process.

It should be noted that array types do not have their own assembly files, so their type label is generated in a special file and contains only a pointer to their column in the subtype table.

It was decided that entire subtype table would be placed in an assembly of its own to achieve modular code.

1.2.7 The String Concatenation Operator +

To generate correct code for the string concatenation expression using the operator +, we tranform the sub-tree of the expression in AST, using the *String.valueOf(...)* methods and the *concat(...)* of *java.lang.String* class. If one operand of the operator is of a primitive type and the other a *String*, the *valueOf()* mehod that corresponds to the primitive type is called to convert the sub-expression in to a String. After making sure both sides are *String*s, we call the *concat* methods of the left operand, and use the right operand as its argument. For example, the expression *"string" + 123* would be transformed into the equivalent explicit expression *"string".concat(String.valueOf(123))* and then the assembly code would be generated for the new expression.

1.2.8 Implicit Conversion in Assignment

Similar to the situation in the last section, Java requires primitive types to be implicitly converted into their corresponding class object. For example the expression *Integer i=1* must result in a *java.lang.Integer* in *i.* To support this feature we use the same strategy as in string concatenation operator. We transform the expression using the constructor of the corresponding class. The above example would be turned into *Integer i = new Integer(1)* before the code is generated.

1.2.9 Literals

Literals are created using the *dd* instruction in NASM, under labels unique to the assembly file. For example, an integer literal 1 would generate code *INT_1: dd 1.* The *null* literal and the *false* boolean literal both have value 0. The *true* boolean literal has the value *0xffffffff.*

String Literals are created using the *dd* instruction too, but the String literal completely follows the structure of a *java.lang.String* object. As shown in Figure 1.3. The actual String value will be stored as an array under the *chars* field. This way the string literal can function as a real object and support method calls.

Chapter 2

Implementation and File Organization

As was mentioned in Section 1.2.1, the code generator is implemented as a visitor. To create better modular code, we separate this process into three specialized classes: *CodeGenerator*, *StatementCodeGenerator*, and *ExpressionCodeGenerator*. The *CodeGenerator* is the top level visitor that generates class skeleton for the assembly file, whereas the *StatementCodeGenerator* and *ExpressionCodeGenerator* as their names suggest are responsible for generating statement and expression code respectively. These three classes are all visitors and the highe level visitor calls the lower level visitors in the code generation process.

The code generated for a specific class are attached to its AST. After the actual code is produced, it would be placed in corresponding assembly files by the *CodePrinter*.

The *Offset* class implements functions for computing instance fields and methods offsets. It also computes the interface method table for interface method invocation. The *VairableOffSetVisitor* is used to compute offsets for formal parameters and local variables. Similarly the *HierarchyTableBuilder* is used to construct the subtype table used for runtime type checking.

In terms of the output, the assembly for each class is stored in an assembly file with the fully qualified name of the class. Several special purpose assembly files are generated. For instance, the *hierarchy.s* file contains the subtype table, and the *staticinit.s* assembly file contains a subroutine that initializes all static fields, which will be called before running the *test()* method in the *_start* subroutine.

Chapter 3

Testing

The assembly code generated by program can be hard to read, and even harder to debug. To ensure the correctness of our code, we followed the unit testing strategy and devised test cases for each scenario. Test cases were created for each expression and statement, and they were made as small and modular as possible. For example, when testing the *if* statement, we created test cases for both when it is true or when it is not. The cases with or without the else statement are also tested separately. This strategy has proved to be very useful because it shows clearly what is working and what is not. It also helped us to quickly identify the source of a problem.

At the machine code level, we need to manipulate memory directly, so even a tiny misconduct may cause the memory corruption which is hard to detect. Therefore we also construct large test cases with different combinations of expressions and statements, in order to check the program thoroughly. These test cases gives us confidence over the robustness of the compiler. Besides, every time a new feature of Java is implemented in x86 assembly or a change is made to the program, we rerun all the previous test cases to ensure that no new bugs are introduced to the program.

The test cases from Marmoset are great resource that can help us test our compiler, so we made a JUnit test suit that runs all the Marmoset test cases locally. We also test the finished compiler with the test cases from previous assignments to ensure the code generation is functioning correctly. The final product has no know bugs, and can pass all the test cases successfully.

On top of unit testing, we also use the *OutputStream.println()* method provided by the stand library to visually check that the result of expressions are correct. For example, arithmetic operations and literals are checked with this method.