Victoria University of Wellington
School of Engineering and Computer Science

SWEN430: Software Development

Assignment 3 — Bytecode Generation

Due: Monday 6th September @ Mid-night

Overview

This project is concerned with compiling the WHILE language to Java Bytecode. To help with this, the `jasm` library is provided for reading and writing Java Bytecodes (see `http://github.com/Whiley/Jasm`). You can download the latest version of the compiler from the SWEN430 homepage, along with the supplementary code associated with this assignment. *We recommend that you start from a fresh version of the compiler, rather than any extensions from previous assignments.*

HINT: see the appendix for tips on debugging!

NOTE: you should not attempt to modify the `jasm` library in anyway, and your program should compile against the stock version provided on the course homepage. Alternatively, you can convert your Eclipse project to a "Maven project" and the `jasm` dependency will be resolved automatically based on the supplied `pom.xml`

NOTE: for the provided test cases to work, you will need to add the directory `tests/valid` as a "Class Folder" in Eclipse. You can do this by choosing "Build Path→Configure Build Path" from the context menu.

1 Fundamentals (20%)

The core part of this assignment is the implementation of the fundamental types, statements and expressions necessary to execute simple programs.

Data Types. The primitive types (`bool`, `int`) should ideally be compiled down to their equivalent JVM primitive types. As a simple example, consider the following WHILE code:

```
1  int sum(int x, int y) { return x + y; }
```

This should compile down to the following Java Bytecode:

```
1  static int sum(int, int);
2   Code:
3     0:   iload_0
4     1:   iload_1
5     2:   iadd
6     3:   ireturn
```

One of the challenges is that every bytecode requires a known type (e.g. `int` in this case). This information can be extracted from the attributes associated with instances of `whilelang.ast.Expr` after the `TypeChecker` has run.

HINT: the logical operators `&&` and `||` should ideally be implemented with *short circuiting* semantics. Whilst one can try to avoid this, it is easy enough to find test cases which fail without this behaviour.

Testing. The main objective for this part is to implement the set of binary operators permitted on the primitive data types above. You will find code for unary operators and various statements, such as **`assert`**, **`return`** and **`if`** is already provided. The test cases in `JvmTests` clarify the set of statements and expressions which must be supported.

2 Complete Control Flow (20%)

Looping statements (i.e. `for`, `while`), can be implemented on the JVM using conditional and unconditional branches. The following illustrates a very simple example:

```
int f(int n) {
   int i = 0;
   while(i < n) {
      i = i + 1;
   }
   return i;
}
```

This should compile down to something similar to the following Java Bytecode:

```
int f(int);
Code:
  0: iconst_0
  1: istore_2
  2: iload_2
  3: iload_1
  4: if_icmpge 14
  7: iload_2
  8: iconst_1
  9: iadd
 10: istore_2
 11: goto   2
 14: iload_2
 15: ireturn
```

Here, we can see the conditional branch at bytecode `4` is used to implement the *loop condition*, whilst the unconditional branch at bytecode `11` returns execution to the top of the loop for the next iteration.

HINT: the **`break`** and **`continue`** statements present one of the main challenges here. The recommended solution is to extend `ClassFileWriter.Context` with appropriate destinations for them (e.g. `breakLabel` and `continueLabel`).

HINT: You are recommended against using `Bytecode.Switch` to implement **`switch`** statements. This will work for integer and related values, but will not subsequently work for other compound types (e.g. records, arrays, etc). A simple and effective alternative is to employ sequences of conditional branches.

Testing. The fundamental statements required for this part are **`while`**, **`do-while`**, **`for`** and **`switch`**. The test cases `JvmTests` clarify the set of statements and expressions which must be supported.

3 Compound Data Types (30%)

The translation of compound types, such as records and/or arrays, presents a number of challenges. Given that performance is not an issue, it is recommended to translate record types into `HashMap`'s, and array types into `ArrayLists`. Unfortunately, this raises the question of handling primitive types (i.e. since neither of these containers will hold a primitive type). The simplest solution is *box* and *unbox* primitive types as they are put into and taken out of such containers. The following illustrates:

```
1  int get(int[] xs, int idx) {
2    return xs[idx];
3  }
```

This is then translated into the following bytecode:

```
1  static int get(java.util.ArrayList, int);
2   Code:
3     0:   aload_0
4     1:   iload_1
5     2:   invokevirtual   #2; // Method java/util/ArrayList.get:(I)LObject;
6     5:   checkcast       #3; // class java/lang/Integer
7     8:   invokevirtual   #4; // Method java/lang/Integer.intValue:()I
8     11:  ireturn
```

Here, we can see that the method `java.lang.Integer.intValue()` is used to perform the unboxing operation.

HINT: You are strongly advised against implementing WHILE arrays using Java arrays. This is difficult if not impossible to make work in general. The primary problem relates to equality of nested array types.

HINT: The methods `boxAsNecessary()`, `unboxAsNecessary()` and `addReadConversion()` are provided to simplify the process of boxing and unboxing primitive data types.

HINT: Value semantics in WHILE represents something of a challenge because compound types (i.e. records and arrays) are implemented on the JVM as references, but this does not reflect their semantic in WHILE. To help with this, the method `cloneAsNecessary()` is provided.

HINT: The method `Collections.nCopies()` is helpful for implement array generators.

Testing. The test cases `JvmTests` clarify the set of statements and expressions which must be supported.

4 Union Types (30%)

The translation of union types, casts and **null** presents significant challenges. In particular, this requires more complex management of implicit coercions between types. The following illustrates:

```
1  void main() { int|null x = 1; }
```

This should be translated as the following bytecode:

```
1 static void main();
2  Code:
3    0: iconst_1
4    1: invokestatic  #16 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
5    4: astore_0
6    5: return
```

Here, we can see an *implicit coercion* has been added to change the representation of `1` into something compatible with the type **int|null** (which is represented on the JVM as an instance of `java.lang.Object`).

Testing. The test cases `JvmTests` clarify the set of statements and expressions which must be supported.

Submission

Your assignment solution should be submitted electronically via the *online submission system*, linked from the course homepage. You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable*. See the following Eclipse tutorials for more on this:

   `http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code*.

3. **The testing mechanism supplied with the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. However, this does not prohibit you from adding new tests. *This is to ensure the automatic marking script can test your code*.

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information*.

Note: Failure to meet these requirements could result in your submission being reject by the submission system and/or zero marks being awarded.

Assessment

Marks for this assignment will be awarded based on the number of passing tests, as determined by the *automated marking system*. **NOTE:** the test cases used for marking will differ from those in the supplementary code provided for this assignment. However, the intention of these hidden tests is that they are largely equivalent to those provided.

Appendix

Debugging test cases which generate verification errors can be challenging and there are a few tools available to help.

Javap Disassembler

You can see the bytecodes in a given `class` file using the `javap` command as follows:

```
> javap -verbose tests/Comments_Valid_1.class
...
public static void main();
  descriptor: ()V
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=0, args_size=0
       0: iconst_1
       1: ifne          12
       4: new           #14       // class java/lang/RuntimeException
       7: dup
       8: invokespecial #2        // Method java/lang/RuntimeException
      11: athrow
      12: return
```

Jasm Verifier

Another option which can be helpful is to enable the Jasm verifier, as this reports more useful error messages than the default JVM verifier. To do this, set `ClassFileWriter.debug` to `true`.

NOTE: this flag is not enabled by default because it causes some tests to fail.