CS444/644 Assignment 1 Technical Report

Ho-Yi Fung, Jianchu Li, Zhiyuan Lin

April 3, 2016

Abstract

bleh

Contents

Chapter 1

Introduction

Chapter 2

Design

2.1 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 on to 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.
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).

Figure 2.1: Stack Frame

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

We choose to push arguments onto 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), however, 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.

2.2 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 **??** shows the structure of the stack frame implemented.

2.3 Class Layout

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

1. Virtual table for the class
2. Class Fields
3. Separate initialization subroutines for instance and class fields
4. Implementation of methods

In the case that a *test()* method is specified in the class. A *_start* subroutine is also generated in the class assembly file.

Chapter 3

Implementation

Chapter 4

Testing

Following the practice during the implementation of the scanner and parser phases, we adopt the unit testing strategy, and utilizes the JUnit 4 testing framework to test our program. The unit testing strategy allows us to ensure that each requirement from the assignments is satisfied. For example, in order to test that all accesses of protected class members are legal, we created test cases with different class hierarchy and access scenario. This way we can make sure the protected access checking works as expected. Both positive and negative test cases, derived from the language specification, were used in this process. For instance, each rule that should be checked by the type checker has its own test cases to verify that the implementation is correct.

A major purpose of the semantic analysis is to ensure that the subject program conforms to the Joos 1W specification semantically. In each step, a large number of checks is performed. Therefore there are a lot of different functionalities/requirements to be tested. To allow for more efficient testing, we combined smaller test cases for different scenarios into three test suites, one for each assignment.

Because the development includes occasional modifications and bug fixes to the previous modules, we decided to test the program incrementally. Once a module is integrated into the compiler or a major change is made, all the previous test suites are run to ensure that no new bugs are introduced to the program, and the changes work as planned.

The test cases from Marmoset were added to our test suites and tested against after each assignment is finished. In each JUnit 4 test suite, the parameterized testing method was used to run the test cases from Marmoset, so that we can make sure the program passes every test case.

The final product has successfully passed all the test cases from our test suites. This includes both the test cases created by the team, and those from Marmoset. This gives the team confidence that the implementation is robust and our design reasonable.

Chapter 5

Conclusion

In this report we discussed the design and implementation of the semantic analysis phase of our compiler. The implementation is complete and successful. All requirements from the assignments are satisfied, and all test cases, both those devised by the team and those obtained from Marmoset, are passed. The analyses are performed in time linear to the size of the ASTs, therefore their running time is acceptable and would work for even very large programs. There are, of course, ways to improve the program. For example, reachability analysis and variable analysis could be combined to save one traversal of the AST. It was a conscious choice to separate the two functions to achieve better modularized code. Besides, this choice does not affect the program's runtime asymptotically.

Bibliography

[1] J. Gosling, *The Java language specification.* Addison-Wesley Professional, 2000.

[2] "The Joos language." `https://www.student.cs.uwaterloo.ca/~cs444/joos.html`. Accessed: 2016-02-03.