210: Compiler Design

Spring 2018

Homework 4

Due: May 3, 2018, 10 a.m.

30 Points

Introduction

The objective for this homework is to build an IA32 back-end for the complete JavaLi language. Your solution will be based on the front-end and on the semantic analyzer constructed in the previous homework. Your task is to build the compiler so that all *correct* JavaLi programs are translated correctly. In this homework efficiency of the generated code *is not an issue*.

The JavaLi compiler developed in Homework 1 handles only a handful of constructs, and the programs written with such constructs are not very interesting. You will extend the compiler to handle all JavaLi constructs that it does not already handle, such as:

- method calls
- field references
- boolean variables and expressions
- `while` statements
- `if`, `if-then-else` statements
- assignment statements
- `new` statements for both objects and arrays
- array dereferences
- cast expressions

You compiler should handle expressions of any size. *Use the stack to store values if you run out of registers (spilling).* For example, when generating code for an expression that requires more registers than available in the target platform. In Homework 1 you handled this case by throwing an exception.

We strongly recommend that you implement dynamic error checking only after you verify that your compiler handles the *full* JavaLi language. Otherwise, even if your compiler handles a large chunk of the language, but not all, we will not be able to run any medium-to large-sized programs using your compiler, and will be forced to give a much lower grade.

The enumeration order of the task does not indicate the order in which you have to solve the homework tasks. Please consult section *Exit codes* for the appropriate way to handle runtime errors and exit codes.

Task 1: Field accesses and method invocations

The compiler must handle method invocations, e.g.:

```
foo(param1, param2);
ref.bar(param1);
int i;
i = compute(param1);
```

and field accesses, e.g.:

```
x = foo.bar;
foo.bar = 5;
```

Variables of basic and reference types are passed by value. If a method has been overridden, then the actual type of the object instance determines the version of the method that is executed. This requires implementing *virtual tables*; strategies for doing this are discussed in the recitation session and in the lecture.

The design of the stack frame is part of this assignment. Please document the design in your program (even if you decide to take an existing design, e.g., the one provided by your host OS). There is no need to be creative - it is perfectly acceptable, and even preferable, to use an existing stack frame design. Just allow the reader (grader) to understand what you are doing.

You are required to detect `null` pointer errors when emitting code for a field dereference or for a method call. Please report the appropriate error code when exiting the program.

Task 2: Casts

Your compiler should generate code to check that downcasts are valid dynamically, similar to Java's handling of downcasts. To handle a cast `(T)e`, where `T` is a type and `e` is some expression, you should:

1. Evaluate the expression `e` to some object `o`
2. Check the type of `o` at runtime: if `o`'s type is a subtype of `T`, then the cast is valid.
3. Otherwise, the invalid downcast should be return the appropriate error code.

Note that checking whether one type is a subtype of another at runtime can be implemented with an helper function written in assembly code. Combine the generated data structures representing the class hierarchy with the helper function to implement the subtyping relationship runtime check. Note that it is legal to downcast (from `Object`) to an array type.

Task 3: Arrays

JavaLi arrays are much simpler than Java arrays: all arrays are subtypes only of Object. You are required to implement the following checks for arrays:

- Array size checking. If an attempt is made to create an array with negative size, report the appropriate error code.
- Array bounds checking. Store the length of the array when it is created. If an attempt is made to read or write beyond the bounds of the array, report the appropriate error code.
- Null pointer checking. If an attempt is made to access an array on a null pointer, report the appropriate error code.

Task 4: New Expressions

Creating new objects and arrays should work as normal. Note that JavaLi has no constructors. You do not need to worry about garbage collection of any kind. To obtain memory, it is recommended that you invoke `calloc()`, however, you can also use other means if you do prefer.

Task 5: Bootstrapping

To start execution, your program should define a `main()` function in your assembly code that executes code equivalent to the following JavaLi:

```
Main m;
m = new Main();
m.main();
```

This means that code begins execution by constructing an instance of the `Main` class and executing its `main()` method, with no arguments. The semantic analyzer should already have verified that the main class and method exist.

Task 6: Create test cases

To test your compiler we provide a few JavaLi programs with the skeleton. Your compiler must be able to produce correct executable code for these programs, but you are also required to create more tests to illustrate that your compiler can handle a wide variety of programs. In addition, it is very helpful if you place a comment at the top of the test file indicating what you are trying to test and also name the file accordingly (e.g., with a short meaningful name). To help the grading of your homework, please prefix the file name with "Err" if you expect the program to fail at runtime or use "Ok" if you expect a normal execution.

Exit codes

If no error occurs, you should always terminate execution by *returning normally from the* `main()` *function with the value* `0`. It is important that you return normally rather than calling `exit()`, because otherwise the `stdout` buffers do not always get flushed, meaning that the testing framework does not notice the full output of your program.

If a dynamic error occurs, you should emit an appropriate error message and invoke the `exit()` function with the corresponding error code. The framework will consult the error code to determine if your code is exiting for a valid reason.

- **Ok**: error code 0, no error
- **Invalid Downcast**: error code 1, cast expression where runtime type is not a subtype of cast type
- **Invalid Array Store**: error code 2, value of wrong type stored into covariant array (*deprecated*)
- **Invalid Array Bounds**: error code 3, load or store into array with invalid index
- **Null pointer**: error code 4, access field, access array or invoke method on null pointer
- **Invalid Array Size**: error code 5, create an array with a negative size
- **Possible Infinite Loop**: error code 6, code runs for too long (*reference solution only*)
- **Division By Zero**: error code 7, division by zero
- **Internal Error**: error code 22, some bug in the interpreter (*reference solution only*)

Note that the reference solution uses an interpreter to generate the `.exec.ref` files. The reference solution interpreter detects error conditions (marked above with *reference solution only*) in addition to those that you have to implemented. It also will only allow programs to execute for a certain, fixed amount of steps to prevent infinite loops: the size should be sufficient for most tests.

Hand-in

Please ensure that your final submission is checked into your team's subversion directory, by the deadline. Any commit after the deadline will not be considered. In addition to implementing the full code generator, please include any tests you created. To make it easier for us to locate the tests you have written, please place them in a subdirectory of `javali_tests` named `HW4_team`. Finally, if you have any comment about your solution, please provide a `README` file in the `HW4` directory of your team's repository.