210: Compiler Design

Fall 2015

Homework 1

Due: Oct 07, 2015, 10:15 a.m.

25 Points

1 Introduction

The objective for this assignment is to produce a simple code generator that is able to translate assignment statements to machine (assembly) code.

The input sources that your compiler must handle are described by a subset of Javali, this semester's programming language. You find a description of the full language at
`http://www.lst.inf.ethz.ch/teaching/lectures/hs15/210/`.
For this assignment, however, you need only handle programs that meet the following restrictions:

- The program consists of a single class, called `Main`, and a single method, called `main`, with no fields.
- This method contains only definitions of `int` variables, `write()` and `writeln()` statements, and assignment statements.
- The left-hand side of an assignment must be a scalar integer variable. The right-hand side may be either the built-in function `read()` or an expression.
- There are no method invocations.
- Expressions are restricted to one of the following:
  - Integer constants.
  - Binary operators + (addition), − (subtraction), and $\star$ (multiplication). Division is optional but highly recommended.
  - Unary operators + and −.
- Note that expressions may appear either as the right-hand side of an assignment, or as the parameter to the built-in `write()` function.
- You do not need to check for semantic errors, such as undeclared variables.

To allow you to work on the backend before you have constructed the earlier phases of your compiler, we provide a compiler skeleton that produces the IR (intermediate representation) for the restricted kind of assignment statements that you must handle. You can obtain this compiler fragment from the `HW1` directory of your team's subversion directory at:
`https://svn.inf.ethz.ch/svn/trg/cd_students/2015hs/teams/<YourTeam>`.
You should use this same subversion repository for your development and for submission.

2 Task description

Your task is to extend the compiler skeleton with a simple code generator for the target machine of your choice. You can follow the basic outline of a code generator presented in class (i.e., use a template-matching approach and a simple register allocator). As we've seen in class, the order of visiting (evaluating) a subtree can have an influence on the number of registers needed. Your *code generator should be economical*, i.e. use the smallest number of registers possible.

Another important part of your task is *generating a set of tests*. We provide a few sample tests with the fragment, but you are encouraged to create more tests to illustrate that your compiler can handle a wide variety of programs.

In the framework, we officially support Linux and OS X. A number of students have successfully used the framework with Windows, typically with Cygwin installed, and you are welcome to do so as well. However, as we do not have access to Windows machines for grading, *you must test with Linux or OS X as well*. For this purpose, please use the predefined string constants we provide as part of the framework for emitting your target language. You can find those string constants in the class `cd.Config.java`.

You are required to keep track of the registers used for values and/or results storage by adding an extra field to that AST node. You should then implement a visitor that read these values and prints the tree to a file.

Required for full credit:

- Implement code generation for all required AST nodes, as listed in the first section.
- Your code must use the smallest number of registers possible.
- Generate additional tests.
- Keep track of registers used by adding an extra field to each node.
- Implement a visitor that reads these nodes above and prints the tree to a file.

You do not need to:

- Implement optimizations.
- Check for syntax or semantic errors.
- Store variables on the stack.
- Handle the case when an expression is too large to be processed entirely in registers, even when evaluated in the optimal order.

If you have extra time, you may wish to implement division and proper stack frames in order to save yourself work on future assignments.

3 Assignment Submission

Please ensure that your final submission is checked into Subversion, at the URL shown previously, by the deadline. In addition to implementing the 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 `HW1_team`. 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. Finally, if there any comments you should have on your solution, please provide a `README` file in the `HW1` directory of your team's subversion directory.