210: Compiler Design

Spring 2018

Homework 1

Due: March 15, 2018, 10 a.m.

25 Points

Introduction

The objective for this homework is to implement a simple code generator that translates assignment statements to machine code for the target platform IA32. The input sources that your compiler must handle are described by a subset of JavaLi, this semester's programming language.

In this homework you are *not required* to handle the full JavaLi's specification. However, your compiler must handle programs under the following restrictions:

- The program consists of a single class, called `Main`, which includes no fields.
- The `Main` class includes a single method, called `main`, which contains only definitions of `int` variables, assignment statements, and `write()` and `writeln()` 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:
  - Integer constants and variables.
  - Binary operators `+` (addition), `-` (subtraction), and `*` (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.
- You do not need to implement optimizations.

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 intermediate representation (IR). You can obtain this compiler fragment from the `HW1` directory of your team's subversion directory.

Task 1: Implement code-generator

For the first task, you have to extend the compiler skeleton with a simple code generator for the target machine that handles the restricted JavaLi described above. You are required to implement a simple code generator that follows a template-matching approach and manages registers using the class `cd.backend.codegen.RegisterManager`.

Your implementation must produce code that uses the smallest number of registers possible when evaluating an expression. Remember that the traversal order of a subtree during compilation can have great influence on the number of registers needed to evaluate an expression. However, your compiler is *not required* to handle the case when an expression is too large to be evauated entirely in registers, i.e., when even the optimal evaluation order requires more than the available number of registers.

Your compiler must be able to generate code for all the AST nodes that are required by the restricted JavaLi. To handle variables you can use the `.DATA` instead of the stack.

Please remember that in the framework, we officially support only Linux. However, if you use the framework on macOS or Windows, please use the predefined string constants defined in the class `cd.Config` for emitting your target language.

Task 2: Create test-cases

To test your compiler we provide a few sample 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.

Hand-in

Please ensure that your final submission is checked into your team's subversion directory, 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 you have any comment about your solution, please provide a `README` file in the `HW1` directory of your team's repository.