Project 5: Extensions to the MiniJava Compiler

Due: Friday, March 9, 12:30 pm.

In this assignment you will make two required extensions to the target code generator. You will also make an extension to the MiniJava compiler of your own choosing.

Part A: Immediate Operands

Modify the x86 target code generator so that it can generate immediate operands for integer constant arguments, rather than moving the integer constant to a register and then using that register. For example, for MiniJava source like `x+1`, instead of generating the following x86 target code:

```
movl %ebi(xoffset), %eax  // load x from stack
movl $1, %ebx             // move constant 1 into register
addl %ebx, %eax           // compute x+1
```

your modified compiler should generate the following:

```
movl %ebi(xoffset), %eax  // load x from stack
addl $1, %eax             // compute x+1
```

Your compiler should be able to perform this optimization whenever the first operand of a 2-argument x86 arithmetic, comparison, or move instruction, or the operand of a push instruction, might be an integer constant. In the current x86 target code generator, this includes whenever the right-hand side of an assignment, the argument of the int-to-double unary operator, the length of an arrayed allocation, the second argument of a non-divide integer binary expression, the second argument of an integer compare expression, and an argument of a call is an integer constant expression.

To perform this optimization, I recommend that you define an alternative version of `codegen`, e.g. `codegenOrImmediate`, which is invoked by methods in `X86Target` for any argument subexpressions that are allowed to be immediate operands. By default, `codegenOrImmediate` just does `codegen`, but for an `ILIntConstantExpr`, `codegenOrImmediate` invokes a special emit operation on the target, e.g. `emitIntConstantOrImmediate`, passing the value of the integer constant. This operation is implemented for the x86 to return a special immediate location, e.g. `X86IntImmediateLocation`, which remembers the constant. (Other targets implement this operation in their own way, to account for that target's treatment of immediate operands. It is always safe for a target to implement `emitIntConstantOrImmediate` just as `emitIntConstant`.) Finally, for those operands that allow immediates as an alternative to registers, the call to the `regOperand` helper can be replaced with a new `regOrIntOperand` helper, which

tests the kind of location passed and invokes either `regOperand` (if the location is an `X86Register`) or `intOperand` (if the location is an `X86IntImmediateLocation`). To help monitor when the optimization is performed, the code for `emitIntConstantOrImmediate` can emit a comment indicating that the regular `emitIntConstant` code was optimized away. (This design, using two versions of codegen and different kinds of result locations, is analogous to how boolean-valued expressions can be code-generated in two different ways, depending on whether the result is being consumed by a conditional branch instruction or not.)

Develop test cases that demonstrate that your optimization is performed when it should be, and not performed when it shouldn't be. (You should also confirm that the existing sample programs continue to run correctly with your optimization enabled.) You can use the `-printCode` option to the MiniJava compiler to print out the assembly code that it produces.

Part B: Improved Register Allocation

Modify the x86 target code generator to eliminate redundant loads from stack locations for variables. For example, for MiniJava source like `x+x`, instead of generating the following x86 target code:

```
movl %ebi(xoffset), %eax  // load x from stack
movl %ebi(xoffset), %ebx  // load x from stack
addl %ebx, %eax           // compute x+x
```

your modified compiler should generate the following:

```
movl %ebi(xoffset), %eax  // load x from stack
movl %eax, %ebx           // copy x from existing register location
addl %ebx, %eax           // compute x+x
```

Likewise, for MiniJava source like `x=...; ...x...`, instead of generating the following x86 target code:

```
movl %eax, %ebi(xoffset)  // store x to stack
movl %ebi(xoffset), %ebx  // load x from stack
```

your modified compiler should generate the following:

```
movl %eax, %ebi(xoffset)  // store x to stack
movl %eax, %ebx           // copy x from existing register location
```

Your modified compiler should be able to replace loads from a variable's home stack location with a register move instruction whenever the variable's value has already been loaded into or stored from a register earlier in the same basic block (or more generally

from the same trace back to the previous label statement or function entry) as long as the register hasn't been reallocated to some other value.

To implement this optimization, I suggest that you add an instance variable to the `X86Target` class that stores a map from stack offsets to register locations. The invariant of this map is that, whenever an offset maps to a register location, then that register holds the same value as the stack at that offset. You can define several helper functions for manipulating this map: `clearStackContents()` which resets the map to the empty map, `forgetStackContents(Location)` which drops any mappings for the given location, `lookupStackContents(int offset)` which returns the register location holding the contents of the stack at the given offset, or null of none, and `recordStackContents(int offset, Location)` which remembers that the given register location holds the contents of the stack at the given offset. These helpers can then be called to manipulate the map at the right places:

- in the function prologue and at label statements: clear the map
- at variable assignments: update the mapping
- at variable reads:
    - first try to generate a move instead of a load by checking whether a mapping exists (and printing a comment if successful)
    - if none exists, generate the load and then record that the result register now holds the loaded stack location
- whenever a register is allocated (in the two `allocateRegister` methods) or overwritten (in the `emitSimpleIntBinop`, `emitSimpleIntUnop`, and `restoreRegisters` methods): forget the mapping, if present

Develop test cases that demonstrate that your optimization is performed when it should be, and not performed when it shouldn't be. (You should also confirm that the existing sample programs continue to run correctly with your optimization enabled.) You can use the `-printCode` option to the MiniJava compiler to print out the assembly code that it produces.

Part C: Your Own Extension

Choose some interesting extension to make to the MiniJava compiler, design and implement it, and develop test cases that demonstrate your new extension. Some possible extensions include the following:

- a new code generation target, e.g. a C source code target, or a MIPS, Sparc, or PowerPC assembly target
- an intraprocedural optimization, e.g. constant propagation and folding, or dead assignment elimination, or real intraprocedural register allocation
- a new MiniJava language feature, e.g. a foreach construct, or labeled break and continue statements, or allowing methods to be marked protected and/or private

with appropriate access checking, or exception throwing and catching, or switch statements, or parameterized types (aka generics)
- a MiniJava library extension, e.g. introducing an Object predefined class, perhaps with some predefined methods that can be inherited by all other classes, perhaps allowing arrays to be subtypes of object, too
- a new runtime facility, e.g. real garbage collection or execution profiling

Only simple extensions are required for the project, but more interesting (and difficult) extensions can earn (modest amounts of) extra credit.

Each project team should talk to Larry or Nathan about their plans for their extension, to make sure that it's not too simple nor too difficult.

You should explain your extension in a separate text file.

Turn in the following:

1. Your new and/or modified source files. Clearly identify any modifications to existing files using comments.
2. The text file explaining your extension.
3. Your test cases that demonstrate the correctness of your extensions
4. A transcript of running your compiler with flags appropriate for demonstrating the correctness of your extensions.

As with the last project, name your root project directory MiniJava, and submit the directory. Put your test programs in the SamplePrograms directory. Zip it up and email to Ben by the deadline.