CSE-504 Programming Assignment #5

Code Generation

Due: Fri., May 4, 2012

Description

In this assignment, you will write the code generator the `Decaf` compiler. For this part, you will provide "`code`" methods for the AST classes. The AST has been augmented to now include type information (`type()` method of the Expressions will give the type of the expression). You are also provided with the object code for all the previous modules and the full set of header files needed to compile your code generator. As usual, you are also provided with a driver program that generates the AST and invokes your type checker.

You'll generate code for the CREAM abstract machine. You can download the source code for CREAM from `http://www.cs.stonybrook.edu/~cram/cream/`. The README file in the distribution gives a (simple) procedure to install an emulator for CREAM on your computer. The manual for CREAM is a part of the distribution (see doc directory). You'll use CREAM v1.2 (the latest available).

Mapping Symbols to Memory

Each method and constructor of a Decaf program will correspond to a sequence of instructions in the generated code. Variables, fields and classes behave differently. Each local variable will be mapped to a location on CREAM's stack; the location will be a fixed offset from the base of the current activation record. Each static field of a class will be located in CREAM's static area. Its location will be a fixed offset from the base of the class in the static area. Each non-static field will be located in CREAM's heap. (Note that there is one instance of a non-static field for each object instance.) The field's location will be a fixed offset from the object's base address. So field and variable symbols are associated with "offsets".

The offsets for all fields and variables are computed and stored in the symbol (entity) table. You can access the offset of a field or variable entity using that entity's `offset()` method.

The mapping of symbols to memory, i.e the offset computation, is already implemented in the code given to you (`Allocate.cc`). Your code generator has to use the offset information that is present in the symbol table.

Strings are special in Decaf: you can have string constants, but cannot use them in any computation. Strings are only used for output. Every string constant is kept in a special part of CREAM's static area. Unlike previous incarnations of the AST, in the current one, a string constant is represented by its offset in the static area. So the "*value()*" of a StringExpression is now an integer representing where the string is stored in the static area.

Code Generation

Statements:

You'll write a method called "`code`" that generates code for statements. The generated code will be a sequence of CREAM instructions and labels written to output stream "`codestream`".

Although the format of how instructions appear in the output stream is irrelevant, it is useful to generate each instruction or label in a line of its own. Each instruction should also end with a semi-colon (e.g. `ildc`, which loads an integer constant, should appear as "`ildc(`n`);`" where n is the constant to be loaded.

Labels are of the form `label`n where n is a number. When a label is used to mark an instruction, it will appear in the code as "`label`n`:`" (note the trailing colon). When a label appears as the target of a branch instruction, it will be without the trailing colon (e.g. `jz(label`n`)`).

You can use the structure `StatementContext` to set the inherited attributes needed to process `break` and `continue` statements. Every statement's `code` method takes the StatementContext object as an argument. Note that the code generated for many statements is independent of the context. The loop statements set up the context for their body statements, while the code for `break` and `continue` statements will depend on their context.

There is one special statement, called NativeStatement, that has been introduced to simplify the way Decaf programs can be interfaced with the outside world (e.g. plain C programs). The idea behind this statement is to provide the name of a native C function. The C function itself may be implemented in a separate file that can be linked into the CREAM emulator. In fact, the input/output methods in Decaf are implemented using this native function call interface. *The code generation method for NativeStatement is already given to you and you need not modify it.*

Expressions:

You'll write a "`rcode`" method for (evaluating the r-value of) each kind of expression. Just like the `code` method of Statements, the `rcode` method should write the generated code to `codefile` stream. The `rcode` method takes no parameters.

The key points to note when generating `rcode` are:

- Type coersion: Note that Decaf permits arithmetic operators that mix integer and floating point values. For instance $1 + 2.5$ is a valid expression, which should evaluate to 3.5. But the CREAM instruction set only has pure integer addition (both operands are integers) or pure floating point addition (both operands are floats). So the compiler should generate code to "lift" integers to floating points whenever necessary. For instance $1 + 2.5$ will be evaluated by first lifting 1 to the equivalent float value 1.0, and then the addition will be performed. Note also that the conversion has to be done at run-time. You may use CREAM's `int2float` instruction to do the conversion.

- Short-circuit code: Boolean expression (AND/OR) should be evaluated using short-circuit code.

- L-Values: Only certain expressions have L-values; these expressions are members of a subclass called LhsExpression. In this subclass, you'll see the declarations for four methods:

  - `lcode`: method to generate code for evaluating the L-value of the expression.

    The L-value of a stack variable (local variable) is its offset in the current activation record. The L-value of field access and array access expressions is a bit more complex.

    Code generation can be simplified by considering the L-values for array access to be a "pair" of values (consecutive elements on stack): the base address of the array, and the index.

    Similarly, the L-value of field access can be treated as a pair: the base address the offset of the field from this base address. The base address of a field access depends on whether this is an access to a `static` field or not. Let `b.x` be a field access expression. If `x` is a non-static field, then `b` corresponds to an object instance. In fact, the R-value of `b` should be a reference to the object instance, and is the base address for this field access. The field `x` will be located a fixed offset from the base address of the object. If `x` is a static field in `b.x`, then `b` will refer to a class, and `b.x` will be placed in the static area of CREAM. The "class number" of `b` gives the base address for this access, and the offset of `x` will be the location of `x` in the static area relative to the base of `b`.

  - `store`: method to generate "store" instructions to store values into addresses specified by (previously computed) l-value.

  - `load`: method to load from an address specified by (previously computed) l-value.

  - `indirect`: a boolean method that returns true if the l-value evaluation results in an indirect access (field/array accesses, which generate a base and offset), or a direct access (local variables).

- Method invocation: For a non-static method, the object (accessed by "this" within the method) will be the 0-th parameter (offset 0 from the base of activation record). The arguments of the method will be placed in subsequent positions, in order, in the activation record. Note that static methods do not operate on a given object, and hence the first argument of a static method will be at offset 0 from the base of the activation record. The other arguments will be in subsequent positions, in order.

- Constructor invocation: When a new instance of an object is created, a new object is first allocated on heap (`newobj()` instruction in CREAM). The `newobj` instruction takes the size of the object to be created as a parameter. This new object is then passed as the 0-th parameter to the constructor. The remaining arguments of the constructor will be placed in subsequent locations in order.

- `this` and `super`. These refer to the current object, and hence correspond to the zero-th parameter.

- New array creation: this is done by using the `newarray` instruction in CREAM. The bounds on the array to be created are the arguments to the `newarray` instruction.

- Auto increment/decrement: These are perhaps the toughest instructions to generate code for. Note that, to evaluate `i++`, we need to first evaluate the r-value of `i`, add 1 to it and store the new value back in `i`; the value of `i++` itself will be the same as the value of `i`. So this means we've to somehow save the old value of `i` so that it is not clobbered by the computation of `i++`. CREAM's `dup` instructions are handy here.

  A more complex issue arises when we do, say `b.x++` or `a[j]++`. To evaluate the r-value of `b.x`, we need the r-value of `b`. In order to store the incremented value back, we also need the l-value of `b.x`,which in turn needs the r-value of `b`. Note that we should not evaluate the r-value of `b` twice. (Consider what should be the result of `a[++i]++`: "`++i`" should be done just once. Note this is not the case if we did `a[++i] = a[++i] + 1`, where `++i` will be done twice.) .

Project Path

In the set of files given to you, you'll see a file called `CodeGen.cc`. That file contains `code` methods for classes, methods, and constructors; for "native" statement; and `rcode` for string expression. Your task is to write `code` methods for all Statements, `rcode` for all Expressions, and the l-value methods (`lcode`, `load`, `store` and `indirect`) for LhsExpressions.

Start with classes containing methods with only assignment statements (and that too with constants). Add different kinds of expressions one by one. At some point, you can introduce other statements, and add code generation methods for those kinds of statements. This way, you can build the code generator step by step, until all the methods are written. Test your code generator for simple expressions and statements first; keep field access, array access, and auto-increment/decrement operations for the end.

In principle, you must be able to complete this assignment by modifying only `CodeGen.cc`. If you need any additional functions, you can add these to that file.

Libraries

Decaf now supports external "native code libraries". A Decaf library consists of two parts: a Decaf interface file, and a native code (C) implementation file. For instance, there is a library that comes with CREAM v1.2 called `stdio`, where `stdio` is a valid Decaf program that defines classes and methods which invoke C functions in `stdio.c`. A library can be used in a Decaf program using the "import" construct: e.g.

```
import stdio
```

The Decaf front end reads the library interface file and incorporates those class definitions into the AST. Other files, driver.cc and Header.cc add include directives to the generated program to incorporate the native-language implementation files for emulation in CREAM. When running "demo", the Decaf compiler, the library interface is first searched for in the current directory; if not found, then the directory specified by environment variable `CREAMLIB` is searched. Usually, `CREAMLIB` should be set to the `lib` directory in CREAM distribution. If `CREAMLIB` is undefined or does not contain the specified interface file, the front end returns with an error message.

Running the Emulator

The driver produces `demo` such that `demo` takes a single argument: a Decaf program file, say `foo.decaf`. The generated code is written then written to file `foo.c`. The code file contains initialization routines and header information needed by CREAM "emulator". We first generate an executable from the generated code using the `stir` script of CREAM (which invokes gcc with appropriate actions):

```
stir -g -o foo foo.c
```

The above command will generate an executable `foo` with debugging information. We can finally "run" the original Decaf program by executing `foo`.

During the development of the code generator, you may need to step through the generated code instruction by instruction or inspect the activations produced by your code. This can be done by "debugging" under gdb as long as the executable was obtained with `-g` option. See CREAM manual for more details.

Available Material

The following material is available in the accompanying `.tgz` archive:

- A `src` directory containing source files for the driver, lexical analyzer, symbol table, AST and Type data structures. This directory also has the template `CodeGen.cc` file that you will edit and submit.
- An `include` directory containing header files for the symbol table and AST data structures.
- An `obj` directory containing object file `TypeCheck.o` for the type checker. This directory also contains an executable "`crdemo`" that you can uses as a reference compiler.

  These objects/executables were creating using GCC v 4.2.1 on Mac OS X. If you cannot directly use the object file/executable and need one for a different machine, let me know.
- A `tests` directory containing sample input files `*.decaf` and the respective output files `*.c`.

  Note that there are several valid instruction sequences (output code) for a given Decaf program. The output files in this directory are just one way to compile the programs; your code generator may give a different sequence, which is fine as long as it faithfully represents the given Decaf program.

In addition, you will need CREAM (see `http://www.cs.stonybrook.edu/~cram/cream/`) to test the generated code.

Deliverables

You need to only submit the modified `CodeGen.cc` file. If you added code to any other file, then archive the entire directory structure (include and src directories) along with the modified Makefile using tar and submit that archive.

Please use the assignment form on Blackboard to upload your submission.