CSE-304 Programming Assignment #4

Code Generation

Due: Fri., Dec 11, 2009

Description

In this assignment, you will write the code generator the `Decaf` compiler. The starting point for this assignment is the compiler front-end, including the parser and scanner modules (HWs 1 and 2), AST builder (HW 3), type checker, symbol table module and a skeleton for the code generation module. You have access to the full source code for these modules and the full set of header files needed to compile your code generator. You are also provided with a main program that invokes the code generator and writes out the generated code to files.

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 v1.1 (the latest available).

The overall structure of the compiler front-end, starting from the AST, is described below. The sections on AST structure, type checking, and memory mapping are for your information. The code generation task, which is the aim of this homework assignment, is described in the section "Code Generation" (Section 4, page 6) below.

1 AST Structure

The AST data structures consist of four main parts:

- **Entities:** Information about program symbols such as classes, methods, fields, variables, etc. These data structures are consist of the symbol table for the compiler.
- **Statements:** Information about statements in the program are stored in these structures.

  Expressions: Information about statements in the program are stored in these structures.

  Types: Information about types in the program are stored in this structure.

Each of these parts is described in detail below.

Entities

The symbol table stores information about each identifier in the program. For AST purposes symbols in the table have additional information stored as objects of class "Entities", in particular, objects of one of the following four classes (subclasses of Entities):

VariableEntity: This class contains information about variables in the input program. Each instance of this class has the following information:

- *name:* Name of the variable entity (inherited from `Entity`).
- *type:* A pointer to the type of this variable.
- *dimensions:* An integer, denoting the number of dimensions of this variable. (Scalar variables have dimension 0; Array variables have dimension > 0).

FieldEntity: This class contains information about the fields in the input program. Each instance of this class has the following information:

- *name:* Name of the field entity (inherited from `Entity`).
- *visibility_flag:* A boolean, indicating whether the field is public or not. I.e. *visibility_flag* is `true` is the field is public, and `false` otherwise.

  Note that the default visibility is public.
- *static_flag:* A boolean, indicating whether the field is a static field or not. I.e. *static_flag* is `true` is the field is static, and `false` otherwise.

  Note that a field is non-static by default.
- *type:* A pointer to the type of this field.
- *dimensions:* An integer, denoting the number of dimensions of this field. (Scalar field have dimension 0; Array fields have dimension > 0).

MethodEntity: This class contains information about the methods in the input program. Each instance of this class has the following information:

- *name:* Name of the method entity (inherited from `Entity`).
- *visibility_flag:* A boolean, indicating whether the method is public or not. I.e. *visibility_flag* is `true` is the method is public, and `false` otherwise.

  Note that the default visibility is public.
- *static_flag:* A boolean, indicating whether the method is a static field or not. I.e. *static_flag* is `true` is the method is static, and `false` otherwise.

  Note that a method is non-static by default.
- *return_type:* A pointer to the return type of this method.
- *formal_params:* A list of entity pointers representing the formal parameters of this method. Note that each formal parameter is an instance of *VariableEntity*.
- *method_body:* A pointer to this method's body statement.

ConstructorEntity: This class contains information about the constructors in the input program. Each instance of this class has the following information:

- *name:* Name of the constructor entity (inherited from `Entity`).
- *visibility_flag:* A boolean, indicating whether the constructor is public or not. I.e. *visibility_flag* is `true` is the constructor is public, and `false` otherwise.

  Note that the default visibility is public.
- *formal_params:* A list of entity pointers representing the formal parameters of this constructor. Note that each formal parameter is an instance of *VariableEntity*.
- *constructor_body:* A pointer to this constructor's body statement.

ClassEntity: This class contains information about the class symbols in the input program. Each instance of this class has the following information:

- *name:* Name of the class entity (inherited from `Entity`)
- *superclass:* Superclass (if any) of this entity. This field points to the superclass entity. If the entity has no super class, then this field is NULL.
- *class_members:* A list of entities representing the members (i.e. fields, methods and constructors) of this class entity.

The fields themselves are private and cannot be directly accessed; use the accessor methods of the same name to get a field's value. For instance, the type of a variable entity `ve` can be accessed as verb+ve-¿type()+.

Some fields may be modifiable after the entity is created; the modification methods are denoted by their `set_` prefix. For instance, to change the type of a variable entity `ve`, to a different type, say `newtype`, you can use `ve->set_type(newtype)`.

In addition, each of these classes have a `print` method which writes out the information to `cout`.

The entity classes are defined in file `AstSymbols.hh`. Scan that header file for a fuller understanding of the API.

Statements:

Statements are a set of classes implementing the following tree grammar:

```
statement:
      IfStatement(expression, statement, statement)
    | WhileStatement(expression, statement)
    | ForStatement(statement, expression, statement, statement)
    | ExprStatement(expression)
    | ReturnStatement(optional expression)
    | BlockStatement(list of statement)
    | DeclStatement(list of entities)
    | BreakStatement
    | ContinueStatement
    | SkipStatement
```

More concretely, `Statement` is an abstract class ("pure virtual") from which classes representing the different kinds of statement are derived. For instance, `IfStatement` is a derived class (i.e. subclass) of `Statement` and has three fields: `expr`, a pointer to an instance of `Expression` class (see "Expressions" below) that denotes the condition in the if-statement; `thenpart`, a pointer to an instance of the `Statement` class that denotes the "then part" of the if-statement; and `elsepart`, a pointer to an instance of `Statement` class that denotes the "else part" of the if-statement. These classes are declared in file `Ast.hh`. As you can see, most of these class definitions are straightforward and the meaning of the various fields are clear from the context. The less obvious statement classes are described below:

- `ForStatement`: There are four parts to a `for` statement. For instance, consider the following code fragment:

  ```
  for(i = 0; i< n; i++)
     sum = sum+a[i];
  ```

  In the above example, `i=0` is called the loop initializer. This is a statement (actually, always a specific form statement called `ExprStatement`). The `init` field of `ForStatement` is a Statement pointer, and represents this component of for-statement. The fragment `i<n` is called the loop guard, and is represented by the `guard` field (an `Expression` pointer). The fragment `i++` updates the loop counter; this part is also in general an ExprStatement, and is represented by `update` field, which is a Statement pointer. Finally, the body of the loop (`sum = sum+a[i];`) is represented by `body` field which is also a Statement pointer.

- `BlockStatement`: When a sequence of statements is enclosed in braces (i.e. between "{" and "}") it forms a block statement. Thus BlockStatement has a single field, `stmt_list` which is a (pointer to) a list of Statement pointers. The order of statements in this list is the same as the order in which they appear in the program text. This list is implemented by the STL container `list`.

- `DeclStatement:` Represents variable declarations. The set of variables declared (i.e. pointers to VariableEntities) is represented by the field `var_list`.
- `ExprStatement:` Represents "statement-like" expressions such as assignments, method invocations, etc. This class has a single field `expr`, which is a pointer to the Expression instance.
- `SkipStatement:` Represents a "NOP" (do-nothing) statement. For instance, an if-statement with no else part is represented in the AST with an `IfStatement` object whose `elsepart` field points to a skip statement.
- `ReturnStatement:` Represents a "return" statement with the expression being returned as the argument. If the return statement is used without a value (e.g. as in a void method), them its argument is `NULL`; otherwise, the argument is a reference to the returned expression.

Expressions:

Expressions are a set of classes implementing the following tree grammar:

```
expression:
    AssignExpression(expression, expression)
    | MethodInvocation(expression, name, sequence of expressions)
    | BinaryExpression(binary_operator, expression, expression)
    | UnaryExpression(unary_operator, expression)
    | AutoExpression(auto_operator, expression)
    | ArrayAccess(expression, expression)
    | FieldAccess(expression, name)
    | NewInstance(name, sequence of expressions)
    | NewArrayInstance(type, name, sequence of expressions)
    | ThisExpression
    | SuperExpression
    | IdExpression(entity)
    | NullExpression
    | IntegerConstant(int)
    | FloatConstant(float)
    | BooleanConstant(bool)
    | StringConstant(string)
```

This tree grammar is implemented as a C++ datastructure in the same way as done for Statements: Expression is an abstract class, and for each kind of expression, there is a derived class. The definition of these classes is also in `Ast.hh`. Again, the fields of these data structures are clear from the context most of the time. The less obvious ones are described below:

- `MethodInvocation:` The first field is a "path" to the method, the object on which this method operates. For instance, in the expression `a.b.f(2)` is a method expression where the prefix `a.b` corresponds to the object for this method, the method name is `f` and its argument is a singleton list `2`. If a method is invoked without a "path" prefix, then it is assumed to operate on `this` object.
- `AutoExpression:` Expressions that do auto increment and decrement (e.g. `i++`) are represented as auto expressions.
- `FieldAccess:` Expressions of the form `a.b.c` are field access expressions. The prefix path (`a.b` in the above example) is the first field (`base`), and the field name (e.g. `c`) is in `name`.
- `NewInstance:` These expressions are used to create new instances of classes (e.g. `new List()`). The class (e.g. `List`, and the arguments to pass to the class constructor are the fields of `NewInstance`.

- `NewArrayInstance`: These expressions are used to create new instances of arrays (e.g. `new int[10][20][]`). The element type (e.g. `int`), dimensions (3 in this example) and the sequence of expressions representing array bounds (10,20 in this example) are the fields of `NewArrayInstance`.

In the AST, any reference to a local variable (e.g. formal parameter or a local variable of a method) will be represented as an `IdExpression`. Any reference to a field will be a `FieldAccess` expression. Note that in the source program, a field expression may look like `b.x` where `b` is the object being accessed and `x` is the field within the object. Note also that the object `b` may be implicit, hence just `x` by itself may refer to a field in the current object. In the AST, implicit object references are made explicit: an implicit reference to field `x` will be treated exactly as though the reference was to `this.x`.

Types:

Types are a set of classes implementing the following tree grammar:

```
type:
     IntType
   | FloatType
   | BooleanType
   | StringType
   | VoidType
   | ClassType(entity)
   | InstanceType(entity)
   | ArrayType(type)
   | UniverseType
   | NullType
   | ErrorType
```

The first four are primitive types, denoting integers, floating point values, boolean values, and string constants respectively. `VoidType` is used to denote the return type of `void` methods. `ClassType` is used to denote the type of class identifiers (e.g., such as those used for invoking static functions or accessing static fields). `InstanceType` is used to denote the type of objects. `ArrrayType` is used to denote types of arrays; the `type` parameter of `ArrayType` is the type of array elements. The last three, `UniverseType`, `NullType` and `ErrorType` are used internally by the type checker.

This tree grammar is implemented in C++ class in the usual way.

2 Type Checking

The type checker module does two things: (a) it decorates the AST with types of expressions (which you can get to by using the `type()` method of `Expression` objects), and (b) resolves overloading and identifies base classes from which methods and fields are inherited from. For the purposes of this assignment, you will not need to know the internals of the type checker.

3 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. 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.

4 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.)

5 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, constructors and statements; `rcode` methods for expressions; and l-value methods (`lcode`, `load`, `store` and `indirect`) for LhsExpressions. `GenCode.cc` currently defines these methods to write "undefined" on to the generated code file. For some methods, the code is fully defined; these will serve as templates or examples to you. For instance, the `code` methods for "If" statement is fully defined. You will write `code` methods for the other statements (except "Decl" and "Skip" statements that generate empty code). Similarly, the code methods for "array access", "unary expression", "id expression", "null expression", "integer constant expression" and "string constant expression" are all fully defined.

So, for this assignment, you have to look for functions with "undefined" in their bodies and fill those functions.

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, method invocations 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 v1.1 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.

6 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.
- A `include` directory containing header files for the symbol table and AST data structures.
- 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.

7 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.