CS164: Programming Assignment 5
Decaf Semantic Analysis and Code Generation

Assigned: Sunday, November 14, 2004
Due: Thursday, Dec 9, 2004, at **11:59pm**
No solution will be accepted after Sunday, Dec 12, 11:59pm

Start early, or even earlier!

December 3, 2004

1 Introduction

Overview

In this assignment, you will complete the implementation of *Decaf*. Building on your solution from PA4, where you implemented a *SkimDecaf* compiler, you will now implement both the semantic analyzer and the code generator for the object-oriented features of *Decaf*.

The assignment has again been broken into several steps, however, you don't need to adhere to these steps religiously . Still, implementing the *Decaf* features in the suggested order will improve your score if you happen to run out of time (PA5 this is the most complex of the five assignments, and, as in PA4, we are giving you virtually no starter kit). The recommended steps are:

1. You will first implement object creation (the `new` operator).
2. Next, you will implement access to instance (i.e., non-static) fields.
3. After that, you should implement instance (i.e., non-static) methods.
4. Next, you should add support for inheritance.
5. Finally, you should add semantic checks to handle incorrect *Decaf* programs.

On this project, you will work in pairs or alone. You are allowed to form different teams than those in previous programming assignments, but you are not allowed to disband a team in the middle of a project assignment. **You must submit a file named PARTNERS, which should list the login names of the both partners.** The PARTNERS file has a specific format, which you must follow exactly. See **Handing in the Assignment** at the end of this document for details. The failure to submit the PARTNERS file will result in a (small) penalty.

Optimania. As promised in PA4 handout, PA5 will include an optional **Optimania** contest in which you can earn extra credits for fast-running code (only if this code is also correct). You will submit your optimizing compiler for the contest as a separate assignment (called PA6) a day or two after PA5 is due (details to follow).

Be careful: Grading of PA5 is going to be very strict as to the *correctness* of the generated code. So, if your optimizer and optimized code generator is not fully debugged, don't submit it as part of your PA5 solution.

Note: To participate in Optimania, we will require that your compiler supports an `input` language construct, which reads an integer value from the standard input (assume that the input will contain exactly one integer per line.) The input "operator" behaves like a call to a static method: `int x; x = input();`.

To implement the input operator, use the C function `scanf`. You need to generate code similar to the x86 code fragment shown below. In case you are curious, the fragment below comes from code generated by gcc from the C program `main() { int temporaryVar; scanf("%d", &temporaryVar); }` .

```
        .section        .rodata
.LC0:
        .string "%d"

        ...

        leal    -4(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    scanf
```

Some explanation is in order. `scanf` takes two arguments: the first is a format string (as in `printf`, which you used to implement *Decaf*'s `print`); the second is an address of a memory location, where `scanf` will store the integer read from the standard input. In the code above, this locations `-4(%ebp)`; you will want to replace `-4(%ebp)` with the address of a temporary generated by your compiler.

What you are going to learn and why it is useful

First, you are going to learn how to generate code for an object-oriented program. In doing so, you will learn why object-oriented programs run slower than their imperative counterparts. This knowledge is useful for each modern programmer, as it allows her to write a more efficient Java code.

Second, you will learn how an object-oriented compiler typechecks the program. This will reinforce your understanding of what kind of programming mistakes the compiler is capable of detecting (at compile time).

2 Getting the Project Files

The starter kit is available on the web at:

`http://www-inst.eecs.berkeley.edu/˜cs164/starters/PA5.zip`

You can also get the starter kit from your Unix account at:

`˜cs164/public_html/starters/PA5.zip`.

Usage of the starter kit is the same as PA1. Please to refer to the PA1 handout for information on how to import the zip file into a new project.

The starter kit is minimal. Essentially, it contains only a file with print statements for the semantic checks.

Note that we are not providing a solution to PA4. Please use the remote tester to fix the bugs in your PA4 solutions. You can find the solutions to PA1–3 in the start kit for PA4.

3 *Decaf* Language Specification

In this programming assignment, you are going to implement a complete compiler for *Decaf*. You will build upon your *SkimDecaf* compiler from PA4. In PA5, you will implement those *Decaf* features that are outside the *SkimDecaf* language.

This section defines the language features that you are to implement in this assignment. First, we outline *Decaf*'s syntactic features that *SkimDecaf* didn't support and hence you have to implement them now. For details of *Decaf* grammar, refer to the *Decaf* grammar in PA3.

- A *Decaf* program may contain multiple classes.
- *Decaf* supports both static and instance fields. Same for methods. Recall that the *Decaf* grammar in PA3 doesn't allow static methods; the support for static methods was added to *SkimDecaf*and *Decaf* in PA4.
- A *Decaf* program may contain the "dot" operator, which is used to access fields and to invoke methods.
- *Decaf* supports the `new T()` operator, which creates a new object of type `T` on the heap. Note that *Decaf* doesn't support constructors.
- *Decaf* supports the `this` keyword.
- The keyword `null` is supported in *Decaf*.

***Decaf* semantics.** We now clarify some semantic properties of *Decaf* programs that are not specified by the *Decaf* grammar. Semantics not stated here is as in Java; when in doubt, ask the course staff.

- As in *SkimDecaf*, a *Decaf* program's start point is the `main` method in class `Program`.
- A class name can be used before it is defined.
- All methods are assumed to be public.
- All fields are assumed to be protected.
- All instance methods have the implicit argument `this`, which takes the value of the receiver object.
- Argument evaluation in method calls proceeds left to right. If you implemented the opposite evaluation order in PA4 (because you push the last argument first), a relatively easy fix is to reverse the order of how you push arguments on the stack (pushing the first argument first is acceptable in *Decaf*, as *Decaf* doesn't support a variable number of arguments).
- A reference to an instance field x of a class T is legal only in instance methods of class T, and is internally translated to *this*.x.
- A reference to an static field x of a class T is legal in both instance and static methods of class T (and its subclasses), and is internally translated to $T.x$.
- The operands of ⟨EQOP⟩s must have conforming types, but cannot be of type String.
- The type of the expression `this` is the lexically enclosing type.
- The null pointer is represented as a memory address with the value 0.
- Object fields are initialized to their default values when an object is constructed using `new`. Object locations (i.e., object references) have a default value of null, integers have default value of zero, and booleans have a default value of false. (Local variable are are initialized, too; see the handout for PA4.)
- Methods (both instance and static) cannot be overloaded. That is, no two methods in the same class can have the same name, regardless of their argument types and return types.
- Instance methods can be overridden. Instance fields cannot be overridden. Static fields and static methods are not inherited.
- If the same static or instance field name is defined in in multiple ancestor classes, the access to that field must refer the field defined in closest ancestor.

```
class A { int x; }
class B extends A {
 static int x;
 void foo() {
  x = 10;  // accesses B's field x
  B.x = 5; // accesses the same field
 }
}
```

Also see Step 5 for semantic checks that you will have to implement.

4 Project Steps

As in PA4, the starter kit contains nearly no skeleton code. The starter kit provides only one class, named `DecafSemanticError`, that you *will have to use* for printing type-checker's error messages. All provided files are described below.

Requirements We are providing you with a `Main` class that should simplify the task of making your compiler run with the remote tester. For the record, what we require from the Main class file is that your compiler compiles a *Decaf* file into an assembly file. Specifically, when your compiler is invoked as

```
java edu.berkeley.cs164.compiler.Main decafpath assemblypath
```

it must compile the file named `decafpath` into a file `assemblypath`. The former will typically have an extension `.decaf`, and the latter `.s`.

In all steps, except the last one, assume that the input is a legal (well-typed) *Decaf* program. The last step will implement semantic checks to ensure that your compiler or generated code doesn't misbehave.

Step 1 – Object creation.

Implement object creation, using the `new` operator.

- To create an object on the heap, use the C function `malloc`, which accepts as its only argument the number of bytes to allocate and returns the address of the allocated block of memory. It is the *Decaf* programmer (not the *Decaf* compiler or runtime) who is responsible for checking whether the return value of `new` is `null`.
- *Decaf* doesn't support memory management (automatic or manual), so you don't need to deallocate garbage objects.

Step 2 – Field access.

Next, you will implement access to instance (i.e., non-static) fields.

To access an instance field of an object, you may want to generate code similar to the following example, which computes the value of `x.field`, assuming that `field` is at byte offset 12 within its object and that `x` resides in memory location `8(%ebp)` (i.e., this memory location contains the address of the object):

```
movl    8(%ebp), %eax
movl    12(%eax), %eax
addl    $13, %eax
```

Step 3 – Non-static method invocation.

Next, you should implement invocation instance (i.e., non-static) methods.

To invoke an instance method of an object, you may want to generate code similar to the following example, which executes `x.f()`, assuming that `f` resides at byte offset 4 in the dispatch table and that `x` resides in memory location `-4(%ebp)`:

```
        // move x to %eax
        movl    -4(%ebp), %eax
        // load the address of the dispatch table into %edx
        movl    (%eax), %edx
        // compute the address of f in the dispatch table
        addl    $4, %edx
        // push the implicit argument (this)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        // get the address of f
        movl    (%edx), %eax
        // call the procedure whose address is stored in %eax
        call    *%eax
```

To generate the dispatch table, use the `.long` directives, as shown in the code below, where `_ZTV3Foo` is a label generated by the gcc compiler to refer to the address of the dispatch table for class Foo (when creating objects of type Foo), and `_ZN3Foo1fEv` and `_ZN3Foo1gEv` are labels generated by the gcc compiler for the instance methods f and g of the class Foo.

```
_ZTV3Foo:
        .long   _ZN3Foo1fEv
        .long   _ZN3Foo1gEv
```

Step 4 – Inheritance.

Next, you should add support for inheritance.

If you implemented the previous three steps correctly, you should have to implement no new functionality here. Just make sure you test your compiler extensively on many (sufficiently) different *Decaf* programs that define subclasses.

Step 5 – Semantic checks.

Finally, you should add semantic checks to handle incorrect *Decaf* programs.

You will have to implement the semantic checks shown below. Recall that the grammar symbols denoted with ⟨SYMBOL⟩ refer to the *Decaf* grammar in the PA3 handout.

A note on *SkimDecaf* semantic checks: Your PA5 compiler will have to perform also the *SkimDecaf* checks listed in the PA4 handout. *Some of the PA4 checks will need to be refined!* In particular, the meaning of "type T conforms to type S" is different in PA5, because *Decaf* is an object-oriented language. We define a type T as conforming to another type S if $T = S$ or if T is a (transitive) subclass of S (PA4 defined conformance as $T = S$).

The *Decaf compile-time* semantic checks (that are not supported by *SkimDecaf*):

1. For all method invocations of the form ⟨simple_expr⟩.⟨id⟩()
   (a) If ⟨simple_expr⟩ is a name of a class T, then ⟨id⟩ must name one of T's static methods.
   (b) If the type of ⟨simple_expr⟩ is some class type, T, then ⟨id⟩ must name one of T's instance methods.
   (c) Otherwise, the expression ⟨simple_expr⟩.⟨id⟩() is not well typed.
2. For all locations of the form ⟨simple_expr⟩.⟨id⟩
   (a) If the type of ⟨simple_expr⟩ is some class type, T, then ⟨id⟩ must name one of T's instance fields.
   (b) If ⟨simple_expr⟩ is a name of a class T, then ⟨id⟩ must name one of T's static fields.
   (c) Otherwise the expression ⟨simple_expr⟩.⟨id⟩ is not well-typed.
3. An instance or static field of class T can be accessed only from methods of T and from methods of subclasses of T (i.e., all fields are protected).
4. `this` can be used (implicitly or explicitly) only in instance methods.
5. An ⟨id⟩ used as a ⟨location⟩ must name a declared variable or field (e.g., it's illegal to use a method's name as a location).
6. In calls to `new T()`, T must be the name of a class.
7. The inheritance graph cannot contain a cycle.
8. The same name cannot be used for an *instance* field in two separate classes when one class is an ancestor of the other in the inheritance graph.
9. If T2 is a subclass of T1 and ⟨id⟩ is an *instance* method name defined in both T1 and T2, then the method must have exactly the same signature in T1 and T2. The method signature consists of the method's return type and the types of arguments, in the appropriate order. Note that if ⟨id⟩ is a *static* method, the signatures in T1 and T2 need not match.
10. The operands of ⟨EQOP⟩s cannot be of type String.

Printing the error messages. To print the error messages detected by your semantic checker, use the provided class `DecafSemanticError`.

The *run-time* semantic check. In *Decaf*, one semantic check will be delayed to run time:

1. The value of a location must be non-null whenever it is dereferenced using the "." operator.

When this run-time errors occurs, your program must output to the standard output a message "Null pointer dereference.", after which the program terminates. Make sure that the message is spelled as shown above. You can terminate the program by calling the C function `exit`.

Handing in the Assignment

The submission will proceed as in the previous assignments. Please check the web site for updates.

Good luck!