CS164: Programming Assignment 4
includes Written Assignment 7!
SkimDecaf Semantic Analysis and Code Generation

Assigned: Monday, October 25, 2003
WA7 Due: Thursday, Nov 4, 2004, at **before class**
PA4 Due: Monday, November 8, 2004, at **11:59 PM**

Start Early!

October 25, 2004

1 Introduction

Overview

In this assignment, you will implement both the semantic analyzer and the code generator for *SkimDecaf*, a non-object-oriented subset of *Decaf*. (You will implement the remaining *Decaf* features in PA5.) At the end of this assignment, you will have a complete compiler for a sufficiently powerful language. If interested, you will be able to compare the performance of your generated code with that of your PA1 interpreter.

Somewhat unusually, you will first implement a (simple) code generator, and only then a semantic analyzer, followed by the full code generator. The purpose of this reordering is to show you where and why semantic analysis is needed.

The assignment has again been broken into several steps. After you get familiar with generating x86 code for *SkimDecaf*'s language features, you will complete this assignment in three steps.

1. You will first implement code generation for those *Decaf* features which don't require any semantic analysis. (In this step, you will assume that your compiler is fed only correct *SkimDecaf* programs.)
2. Next, you will add semantic analysis needed to generate code for the remaining *SkimDecaf* features. (You will still assume correct input programs.)
3. Finally, you will implement semantic checks that are needed to ensure that a semantically illegal *SkimDecaf* program doesn't (i) crash your compiler; (ii) make your compiler generate incorrect code; or (iii) cause an error in the assembler or the linker.

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. Handling in will be done as in the previous paojects; watch the website for updates. The failure to submit the PARTNERS file will result in a (small) penalty.

This assignment differs from the previous three in two ways:

- **Written Assignment 6**: While working on this programming assignment, you will have to work on WA7, which is **due earlier than PA4!** In WA7, you will have to answer ten (10) questions about the design of your compiler. You can find these questions in Step 1 and Step 3 of this handout. The assignment will be turned in as a hard-copy as usual.
- **The starter kit.** The starter kit is minimal. Besides examples of how to generate x86 code, the starter kit will contain only one class, with methods for printing the various error messages from the semantic checker.

Optimania. As mentioned above, at the end of PA4, you'll be able to compare the performance of your code with that of your interpreter. This benchmarking is optional in PA4, but it may motivate you to implement a better code generator and/or an optimizer. Producing an efficient code in PA4 may pay off in PA5, which will include an **Optimania** contest in which you can earn extra credits for fast-running code (only if this code is also correct). If you are interested in participating in Optimania, you may want to start implementing optimizations already in PA4. **Be careful, though:** Grading of PA4 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 PA4 solution.

What you are going to learn and why it is useful

First, you are going to learn how to generate assembly code for a procedural language. In doing so, you will learn that a simple code generator is very inefficient, and you will quite likely spot many opportunities for optimizations of the generated code.

Second, you will learn hands-on what facts the semantic analysis collects, and why the code generator cannot (in many case) do its job without the semantic analyzer.

2 Getting the Project Files

The starter kit is available on the web at:

```
http://www-inst.eecs.berkeley.edu/~cs164/starters/PA4.zip
```

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

```
~cs164/public_html/starters/PA4.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 solution to PA3 (which includes the lexer) is available at:

```
http://www-inst.eecs.berkeley.edu/~cs164/starters/PA3solution.jar
```

We are distributing our solution as a jar file containing all the necessary classes, as well as the source code. Do the following to get the jar imported into your project:

1. Download the jar file to your PA4 project directory. Refresh the project in Eclipse so it sees the jar file.
2. In Eclipse, right-click on the project and go to Properties, then Java Build Path in the dialog, and then the Libraries tab. Click the Add JARs button, and choose the `PA3solution.jar` file.
3. Right-click on `PA3solution.jar` in the Package Explorer, and then go to Properties, then Java Source Attachment. Click the Workspace button, and select `PA3solution.jar` again. Now you should be able to view the available source code in the jar.
4. See the source of `Main` in the `edu.berkeley.cs164.parser` package to see how to invoke the parser.

3 *SkimDecaf* Language Specification

In this project, you are going to implement a complete compiler for a subset of *Decaf*. This subset is similar to *SkimDecaf*, the language of PA1, **but it's not identical**. Still, we are going to refer to the language of PA4 as *SkimDecaf*.

This section lists the language features that you are to implement in this assignment. Further clarifications and requirements are given in Section 4. Read that section carefully.

As you will notice, *SkimDecaf* contains no "object-oriented" language features of *Decaf*, such as object fields and inheritance. You will implement those in PA5.

SkimDecaf is described in two steps. First, we constrain *Decaf* into *SkimDecaf* by listing *Decaf*'s syntactic features that *SkimDecaf* does **not** support. In doing this, we refer to the *Decaf* grammar in PA3. Obviously, you are to implement only those language features that are in *SkimDecaf* grammar. Note, however, that the PA3 parser provided by us parses the entire *Decaf* language, and so it will not alert you when your input programs contain a non-*SkimDecaf* feature.

Second, we list semantic properties of *SkimDecaf* programs that are not specified by the *SkimDecaf* grammar.

***SkimDecaf* syntax.** The *SkimDecaf* language is a subset of *Decaf*. Specifically, *SkimDecaf*'s grammar is identical to the *Decaf* grammar in PA3, except for the following:

- A legal *SkimDecaf* program contains only one class definition. This class is named `Program`. Clearly, the `Program` class cannot extend another class, so the keyword `extends` is not supported.

- All fields in a class definition must be `static`.
- All methods in a class definition must be `static`.
- A *SkimDecaf* program may not contain the "dot" operator. That is, a method call consists only of the method name, and a ⟨SIMPLEEXPR⟩ must be an ⟨ID⟩.
- *SkimDecaf* does not support the `new` keywords. That is, a *SkimDecaf* program cannot create non-static objects (i.e., objects that reside in the heap).
- *SkimDecaf* does not support the `this` keyword. As a result, a field may not have the same name as a formal argument or a local variable, otherwise, you would never be able to access those fields, since their name would be hidden.
- The keyword `null` is not supported.

***SkimDecaf* semantics:**

- Each method must end with a return statement. Specifically, the last statement of the outermost block in the method body must be a return statement.
- The class `Program` must contain a method `main`, which is the program's entry point. This method must be **static**, **void**, and have no arguments.
- The types that can appear in declarations are `int`, `boolean`, and `String`. The only exception is the type `Program`, which may appear as the name of the only *SkimDecaf* class definition.
- The `print` statement has one argument which can be of type `int`, `boolean`, or `String`. Except for boolean arguments, the print statement behaves as expected: it prints the values of the integer, or the content of the string (without the surrounding quotes), respectively. Note that this is an exception to the "no overloading rule" above, but it's OK because print will be a bulit in method, so you can handle it differently from regular methods. For boolean arguments, the print statement prints `1` for a `true` argument, and `0` for a `false` argument.
- The type of a string literal is `String`. *SkimDecaf* provides exactly one operation on strings: the print statement. Other operations, such as equality comparison or concatenation, are not supported. However, strings may be used in fields, method call arguments, and method call return values. The run-time representation of strings is up to you. It's probably a good idea to follow the string code generation scheme of the gcc compiler (see below).

4 Project Steps

In this programming assignment, the starter kit contains nearly no skeleton code. The starter kit provides only one class, named `SemanticError`, that you *will have to use* for printing type-checker's error messages. We also provide a few example files on how to generate assembly code. All provided files are described below.

Requirements In contrast to previous programming assignments, we are only providing a very bare-bones starter kit. Since you will have to write a fully functional compiler, all we require that your compiler compiles a *SkimDecaf* file into an assembly file. Specifically, when your compiler is invoked as follows

```
java Main inputfile outputfile
```

it must compile the file named `inputfile` into a file `outputfile`.

Step 0 – Getting familiar with the x86 assembly language.

In this step, you are going to get familiar with the target language of your compiler, the x86 assembler. You will also learn tools for compiling and debugging x86 assembler programs. In this step, you will write no code for your compiler yet, and you will not answer any WA7 questions.

You don't need to *master* the x86 assembly language; you only need to know enough to write a code generator that will translate your *SkimDecaf* programs into x86 programs that will compile and run *correctly*. Since your code generator will produce a simple code, you will be able to get by with knowing maybe two dozen assembly instructions and directives. Furthermore, you will be able to see what assembly code to generate by translating your writing your *Decaf* programs into C (an easy task even if you don't know C) and then translating the C programs to x86 assembly with a C compiler.

x86 Machines. The instructional Solaris/SPARC machines on which you have been developing your compilers (e.g., solar.cs and cory.cs) run on SPARC processors, and so they cannot execute the x86 code that your compiler will generate. To compile and run your x86 code, you will have to log in to one of the many instructional Solaris/x86 machines:

> coralsea.cs, decatur.cs, triton.cs, trenton.cs, johnfk.cs, iwojima.cs, intrepid.cs, hornet.cs, halsey.cs, fulton.cs, johnpaul.cs, leahy.cs, lincoln.cs, nautilus.cs, nimitz.cs, merrimac.cs, skipjack.cs, saratoga.cs, ruebenj.cs, obannon.cs, midway.cs, monitor.cs, somers.cs, tarawa.cs, angeles.cs, america.cs, belknap.cs, brooke.cs, chasseur.cs, clermont.cs.

The need to compile and run the generated code on a different machine should not complicate your development process: all you need to do is to create a terminal (shell) window in which you remotely login to an x86 machine, using ssh; since the Solaris/SPARC machine that runs your Eclipse process and the Solaris/x86 machine share the file system, you won't have to ftp the generated code to your Solaris/x86 machine. If you are working from home, then you will have to use sftp to move your code to the Solaris/x86 machines (unless you have a Solaris/x86 machine at home).

Note regarding NFS. The instructional machines use NFS to share your files across different machines. One issue with NFS is that after you save a file on one machine, there is occasionally a delay of several seconds before the save reaches other machines. During this time, if you try to read or execute the file on a second machine, it may read the old version.

If you want to develop your compiler on your home Windows/x86 or Linux/x86 machine, see the **Be Careful** notes below.

Compiling and running x86 programs. Your generated x86 programs will have the file extension `.s`. The starter kit contains an example x86 program in file `sample.s`. This program shows one possible way of translating the provided *SkimDecaf* program `simple.decaf` into x86.

To compile an assembly program into an executable format, you will use the C compiler `gcc`. Execute the compiler from a shell as follows:

```
gcc simple.s
```

This command will create the executable file named `a.out` (`a.exe` on Windows).

Normally, the C compiler compiles a C program into an executable program. In the command shown above, the C compiler recognizes from the file extension that the argument file is already in an assembly form, and so the compiler merely assembles it into an executable file. The compiler also links your assembly file with C libraries. We will use the C library function `printf` to implement the *SkimDecaf* statement `print`.

To run the executable *SkimDecaf* program produced by gcc, run from a shell window

```
./a.out
```

Debugging x86 programs. To debug your code generator, you may want to run your executable program from the `gdb` debugger. From this debugger, you can single-step machine instructions of the program, print the executable program in disassembled format, as well as examine the contents of machine registers. You will find the following gdb commands handy:

- To start the debugger, run the following command from a shell window: `gdb ./a.out`. This will start the debugger and load into it your executable program.
- To run the executable program from within the debugger, use the gdb command `run`. After the program terminates, you can run it again. Just enter `run` again.
- Before you run the program, you may want to set a breakpoint at the entry point of a procedure. Use the gdb command `break main` where `main` could be the name of any procedure.
- To single-step the execution (instruction by instruction), use the `stepi` command.
- To print the executable code in a disassembled format, use the gdb command `x`. For example, `x /10i main` will print 10 instructions starting at the address with the label `main`, which is where the code of the method `main` starts. Pressing ENTER will print the following 10 instructions.

  After you executed the `x` command with the `/...i` format argument, you can run `x` without the format. For example, you can print the instruction at the beginning of `main` with `x main`. The print the next instruction, just press ENTER.
- You can print registers much like you print variables. For example, to print eax, use `print $eax`. To print 8 words with addresses just before esp, use `x/8x $esp-32` (examine 8 hexwords at `$esp-32`; use `help x` for more information on this command). To print all registers, use `info reg`.

- To learn more about these and other gdb commands, use the handy command `help`.
- There are many useful gdb tutorials on the internet. Be aware, however, that you will be using almost exclusively the *advanced* features of gdb — those designed for debugging *machine* code, at the instruction level. In contrast, the typical use of gdb is to debug C/C++ code, at the statement level. Available tutorials reflect the typical usage. One tutorial that does cover machine-code debugging is at

  `http://www.unknownroad.com/rtfm/gdbtut/gdbadvanced.html`

Examining generated x86 code. To determine what assembly code to generate for a particular *SkimDecaf* construct, you can again use the C compiler `gcc`. All you need to do is to write an equivalent of your *SkimDecaf* program in the C language, and use `gcc` to translate the C program into assembly using the following command:

```
gcc -S -O0 test.c
```

The `-S` option forces a compilation of the C program in the file `test.c` into assembly file `test.s`, as opposed to the executable code `a.out`. The `-O0` option (spelled "o-zero") turns off optimizations, so that gcc generates a *simple* code, giving you examples of what code you may want to generate in your compiler.

Note that the code related to the activation record in `main` may differ from that code in other functions, which is to say that the code generator may need to handle `main` differently.

Writing C-variants of your *SkimDecaf* programs is not difficult (doing the same for *Decaf* programs in PA5 is going to be harder). See for example `simple.decaf` and the equivalent `simple.c` in the starter kit. The internet offers several sites comparing Java and C. If you need more than that, a nice tutorial on C (in Postscript format) can be found at

`http://www.cs.mun.ca/~paul/cs4751/material/c/cfj-two-page.ps`

Developing your code on your home Windows/x86 machine. If you want to develop your *Decaf* compiler on your home Windows/x86 machine, use Cygwin. Cygwin is "a Linux-like environment for Windows." When downloading Cygwin onto your machine, download and install also gcc and gdb, which come as part of cygwin. Cygwin can be found on `www.cygwin.com`.

Be careful: gcc for Windows/x86 generates a slightly different assembly code than gcc for Solaris/x86 (this is because activation record formats and naming conventions differ slightly between Solaris and Windows). Consequently, after developing your *Decaf* compiler on Windows, you will have to port your code generator to Solaris (because this is where we'll grade your compiler and do remote testing). With good software design, porting your code generator should be relatively straightforward.

A sample of changes that you'll need to make when porting from Windows to Solaris (find these changes by comparing the output of `gcc -S` on the two operating systems):

- Different pragmas (i.e., assembler directives) for introducing procedure names and for allocating storage for static fields (which are represented as global variables in C).

- On Windows, code generator prepends the '_' character to procedure names.

Developing your code on your home Linux/x86 machine. You can also develop your compiler on a Linux machine. In the case of Linux, installing gcc and gdb should be unnecessary as these two tools typically come pre-installed with Linux. Again, you'll need to port your code generator to Solaris/x86, as this is where we'll be doing all the grading. *Test your ported code generator thoroughly!*

Classes to Modify:

None.

Other Files Needed:

`simple.decaf`	A simple *SkimDecaf* program.
`simple.c`	The same program translated (by hand) into C.
`simple.s`	One alternative of what your compiler could generate when compiling `simple.decaf` into x86. This file has been generated by `gcc -S -O0`.
`simple-annotated.s`	The same file, but annotated by us so that you can understand which x86 instructions implement which *SkimDecaf* features. See also the solution to WA5.

Step 1 – Code Generation without Semantic Analysis

In this step, you will implement code generation for those constructs of *SkimDecaf* that do not require any semantic analysis. The pedagogical purpose of implementing (some) code generation before semantic analysis is two-fold: (1) you will obtain a running compiler for an interesting subset of *SkimDecaf* pretty soon; and (2) you will learn which language features can be code-generated without semantic analysis, which cannot, and why this is so.

Specifically, your task is to determine which language features described in Section 3 of this handout can be *correctly implemented* without *semantic analysis*.

In other words, you are trying to identify a subset of *SkimDecaf* such that you can take a program that is *correct* and *legal* in this subset of *SkimDecaf* and your compiler with no semantic analysis will be able to produce correct code for it. It is OK for your compiler to crash or generate bad code for a an input program that either has a type error, or which in some way fails to adhere to the subset of *SkimDecaf* you are defining.

What do we mean by "no semantic analysis"? This restriction means that your compiler in this step will maintain no symbol tables, and will propagate no types through the AST. The code generator is, of course, allowed to visit declaration AST nodes and generate code for these nodes. In other words, when you are emiting code for a statement, you will have to make some assumptions about the types of variables in that statement, and you won't be able to distinguish one variable from another.

Try to implement as many features of Skim Decaf as you can without semantic analysis. You will earn full credit for this part if you identify and correctly implement a maximal subset of Skim

Decaf, that is, if there is no way to add more features without semantic analysis. Note that different teams may identify different maximal subsets. You will be graded on this step by your answers in WA7 so please be thorough.

WA7: Answer the following questions. Justify your choice of *SkimDecaf* features that you are able to implement without semantic analysis. Specifically, answer the following questions. For each question, explain the reasons why a particular feature could be implemented without semantic analysis. The answer to each bullet should be at most two-sentences long.

1. Which subset of types `int`, `boolean`, `String` does your semantic-analysis-free compiler support in expression evaluation?
2. Which subset of types `int`, `boolean`, `String` does your semantic-analysis-free compiler support in in the print statement?
3. Are you supporting procedure calls?
4. If yes, what are the limitations on your method calls? (Can calls have arguments? Can they return values?)
5. What subset of *SkimDecaf* statements can you support?
6. Which of the following can you support (static fields, variables)?
7. Give an example of a feature that you could not support without semantic analysis, and explain why.

Step 2 – Semantic Analysis for Code Generation

In this step, you will add that part of semantic analysis that's needed to generate code for the remaining *SkimDecaf* features. Like in Step 1, you will assume that your compiler is fed only correct *SkimDecaf* programs. That is, the input programs will contain no type error from which your compiler needs to recover. Consequently, you will implement no error checking (and print no error messages) in this step.

Hints:

- Propagating type information in AST nodes. You may recall that the visitor implemented in the AST hierarchy doesn't support returning a value other than a boolean. In order to propagate the type information during type checking, you can use the same trick as in PA1.
- Storing type information in AST nodes. To store the type of the AST node directly in this node, you can use the methods `setProperty` and `getProperty` of the class `ASTNode`.

Step 3 – Full Semantic Analysis

In this step, you will complete the *SkimDecaf* compiler by implementing semantic checks that discover semantic errors in incorrect *SkimDecaf* programs,

First, you are going to write input *SkimDecaf* programs that crash your compiler, the assembler, and/or your generated code. These programs will convince you that a decent semantic analysis is useful, and show you what kind of bugs you can find with a semantic checker.

You will then implement the semantic analysis.

Note: Correct implementation of the semantic checks constitutes a significant part of the grade for this programming assignment.

WA7: Answer the following questions. For the following three questions, consider a compiler that was built assuming that it would ever see only legal *SkimDecaf* programs (i.e., programs with no type errors, no missing declarations, no double declarations).

1. Write a short *SkimDecaf* program that makes your *SkimDecaf* compiler misbehave (crash, or terminate with an exception). Explain the reason for the crash.
2. Write a short *SkimDecaf* program that makes your assembler (gcc) print an error message. Explain how it is possible that the error was propagated all the way to the assembler.
3. Write a short *SkimDecaf* program for which your compiler generates incorrect code. That is, the generated code either crashes or otherwise computes an incorrect result.

Semantic Checks. You will have to implement the semantic checks shown below. Note that the grammar symbols denoted with ⟨SYMBOL⟩ refer to the *Decaf* grammar in the PA3 handout. Also note that since PA4 does not deal with any object-oriented features (in particular inheritance), the phrase "the type T_1 conforms to the type T_2" should be interpreted in PA4 as "the type T_1 equals the type T_2". You will implement type conformance in PA5, which will deal with inheritance.

The semantic checks:

1. No identifier is declared twice in the same scope. This applies to method declarations as well. i.e. no overloading; you can not declare two methods with the same name, even if they have different parameters. However, method names are in a separate namespace from fields, locals and parameters, so having a field with the same name as a method IS allowed.
2. No identifier is used unless it has been previously declared in an enclosing scope. In the case of methods, you can use them before they are defined, but you can't use a non-existent method.
3. The program must contain exactly one class called `Program`. This class has a static method called `main` that has no parameters and a void return type.
4. The number of arguments in a method call must be the same as the number of formals, and the types of arguments in a method call must conform to the types of the formals.
5. If a method call is used as an expression, the method must return a result.
6. If a method is declared to return void, all return statements in it must not have an expression.
7. If a method is declared to return a non-void type, all return statements in it must have an expression that conforms to the declared type of the method.

8. The expression in `if` and `while` must have type `boolean`.
9. The operands of ⟨ARITHOP⟩s, ⟨RELOP⟩s and the unary minus operator (−) must have type `int`.
10. The operands of ⟨EQOP⟩s must have conforming types (i.e., either the first operant conforms with the former or vice versa).
11. The operands of ⟨CONDOP⟩s and the logical not (!) must have type `boolean`.
12. The type of ⟨EXPR⟩s in an assignment must conform to the type of ⟨LOCATION⟩s.
13. The procedure must end with a return statement.

Printing the error messages. To print the error messages detected by your semantic checker, use the provided class `SemanticError`. For input programs with multiple errors, you are required to report at least one error, but you are not required to print more than one error. When passing arguments for the SemanticError, please pass null for the ErrorPosition.

Handing in the Assignment

You will hand in this assignment the same way you did PA3.

Good luck!