CS164: Programming Assignment 1
SkimDecaf Interpreter

Assigned: Wednesday, September 1, 2004
Due: Thursday, September 9, 2004, at **noon**

August 28, 2004

1 Introduction

Overview

In this assignment, you will write an interpreter for *SkimDecaf*, a small subset of the *Decaf* language, which is itself a subset of Java. *Decaf* is the programming language for which you are going to write a compiler in this course.

In this assignment, you will, for the first time, implement a programming language (ok, if you wrote a Scheme interpreter in CS61A, then this is already your second language). This interpreter will give you the satisfaction of executing program in your own execution environment, and will prepare you for writing a compiler in the rest of the semester.

Also, you will get practice with tools you will need during the semester: Java, Eclipse, CVS, design patterns, and our remote testing infrastructure.

On this project, you will work alone. On future project assignments, you will work in pairs.

What you are going to learn

First, you are going to learn how programs are represented in a compiler or interpreter. As a programmer, you think of programs as the text-based source code. Programs are entered into the compiler as text because text files are programmer-friendly. They are not compiler-friendly, though. In compilers and interpreters, programs are represented as abstract syntax trees (ASTs), because on these trees we can formulate program interpenetration and translation as convenient tree traversals.

If programs expressed as ASTs are not programmer-friendly, how will you enter them into the compiler? Using a parser. In this assignment, you will translate *SkimDecaf* programs (source text) into ASTs using a parser built in the Eclipse IDE. You will develop a similar parser yourself in PA2 and PA3.

Second, you will also write a pretty-printer for ASTs. The pretty-printer takes a *SkimDecaf* AST as input and prints the program as *SkimDecaf* source text. The pretty-printer can be viewed as a

translator from ASTs to source text. This may sound difficult, but all the pretty printer will do is traverse the AST and print each statement and expression as it reaches it.

Third, you are going to write an interpreter for SkimDecaf ASTs. The interpreter takes a *SkimDecaf* AST as input and runs it. This may sound difficult, but an interpreter is a surprisingly simple program. An interpreter traverses the AST, executing each statement and expression as it reaches it.

The interpreter is based on a traversal of the AST, just like the interpreter, but the traversal order is different. For example, the interpreter may traverse the body of a while loop several times, or zero times, based on the loop exit condition, but the pretty-printer will traverse the loop body exactly once.

Fourth, you will get familiar with the tools you'll need in the rest of the semester:

- *Java and Eclipse.* All programming will be done in Java using the Eclipse IDE. Eclipse runs on many platforms, including Windows, Mac OS, Linux, Solaris, and so you can run Eclipse both in the lab and at home.
- *ASTs.* Rather than rolling our own ASTs, we will use the AST library that comes with Eclipse, in its Java Development Toolkit (JDT). Eclipse uses these ASTs for things like syntax error highlighting and name completion. By relying on these ASTs, you will see a little bit of how a well-designed industrial compiler looks like inside.
- *Design Patterns.* We will use one design pattern (visitor) extensively this semester. In this asignment, you will learn how to program AST traversals with this pattern.
- *CVS.* We will use CVS as our version control system. It will be used for submitting assignments, for coding collaboration within your team, and for remote testing.
- *Remote testing.* To help you find bugs in your project solutions (and thus to score more points on your assignments), we will allow you to compare your solutions against our reference solution, by running test inputs on our solution.

2 The Assignment

Your goal in PA1 is to

1. implement a pretty-printer for *SkimDecaf*,
2. implement an interpreter for *SkimDecaf*, and
3. develop a sufficient collection of test cases to test the pretty-printer and the interpreter.

The interpreter must implement the language according to the specification given in Section 2.1. The pretty-printer must follow the rules in Section 2.2. How your code and test cases must be organized and submitted is described in Section 4.

2.1 *SkimDecaf* Language Definition

SkimDecaf is a subset of Java that supports only integer variables, arithmetic expressions, assignments, `if` statements, and `while` statements. (When reading the details that follow it's useful to recall the difference between expressions and statements.)

Data Types. *SkimDecaf* has only one data type, 32-bit signed integers. (You can implement this data type using Java's `int`.)

Variables. There are no declarations in *SkimDecaf* (that is, both `int x;` and `int x = 3;` are illegal in *SkimDecaf*. All variables have global scope. Programs can use any variable name at any time. It is as if all possible variables are declared as global `int`s.

When a variable is evaluated, the result is the current value of that variable. If the variable has not yet been assigned, the interpreter sets the variable to 1, returns a result of 1, and prints a warning message:

```
Warning:  variable 'x' has not been assigned
```

substituting the name of the variable for **x**. For purposes of this rule, assigning to a variable does not count as evaluating it. For example, if the first statement in a program is `x = 3;`, this does not generate a warning.

Assignment. The assignment expression assigns a new value to a variable, just as in Java. *SkimDecaf* does not support any of the compound assignments such as `+=`. The left-hand side of an assignment operator is always a variable name, which is a `SimpleNamep` AST node.

The result of evaluating an assignment expression is the value of the right-hand side. The following line of code sets `x` to 3, then prints 3:

```
print(x = 3);
```

Print. *SkimDecaf* does not support method calls, except for a call to a built-in method `print` that prints the decimal representation of its single argument, followed by a newline. You will implement `print` by invoking Java's `System.out.println(int)`. The syntax of `print` is as if `print` were a Java method declared like this:

```
public static void print(int x);
```

Note that since `print` is a method call that does not return a value, it must not be nested inside another expression. Compilers normally include a semantic check phase which would detect this problem when the program is compiled. However, interpreters, such as the one in PA1, often do not detect these errors until the program actually does something illegal. So, if a program attempts to use the value of a print expression, the interpreter must throw a `VoidValueException`. For example, the following code would result in a `VoidValueException`:

```
a = 3 + print(10);
```

The exception `VoidValueException` is defined by the interpreter, and already exists in the starter kit. This exception will be caught by `PA1.Main` which is also provided.

Arithmetic. *SkimDecaf* has four binary arithmetic operators: addition, subtraction, multiplication, and division. When evaluating these operators, evaluate the left operand, then the right operand, then apply the operator.

If the program attempts to divide by 0, the interpreter throws a `DivideByZeroException`. The exception `DivideByZeroException` is defined by the interpreter. This exception is different from the exception thrown by Java when the program divides by zero; this exception will be caught in `PA1.Main()`.

Statements. The `if` and `while` statements are just like their Java counterparts, except that the conditional expressions evaluate to integers. Zero represents false, and any other value represents true. The `if` statement optionally includes an `else` clause.

Also, any expression can be a statement. In particular, assignment and print expressions are often used as statements.

Programs. A *SkimDecaf* program contains one class declaration and one method declaration (the method must be static because *SkimDecaf* cannot create objects). It's like a Java class with a main method, and no other methods. Since there is only one class with one method, and the interpreter simply runs the method, the interpreter will ignore the class name and the method name. The names are still required by the language syntax, though. This makes the programs look more like Java, and it will allow us to add multiple methods and method calls in later assignments.

Example. Consider the following (very poorly formatted) *SkimDecaf* program.

```
 public class MainClass {
public static void Main() {
a = 10;
while (a) { a = a-1; print(a); }
}
}
```

Figure 1 (middle column) shows how the pretty-printer formats this *SkimDecaf* program. Note that the pretty-printer fully parenthesizes arithmetic expressions. This is because the structure of the AST determines the order of operations, and the pretty-printer indicates the order given in the AST with parentheses. Some parentheses are clearly unnecessary, but we print all of them to keep the pretty-printer simple.

Next, Figure 1 (left) shows the complete AST for this program, as output by the `ASTPrinter` class included in the starter kit. The class prints out ASTs one node per line. Each line shows the name of the class of the AST node. Lines for `SimpleName` nodes also show the identifier for the name. Lines for `NumberLiteral` nodes show the exact text of the literal.

For illustration, Figure 2 shows an *informal* graphical representation of this AST (the graphical view omits some nodes; compare it with the complete AST in Figure 1). You don't need to print the AST in this project; print it only to understand the AST layout. To print the AST, you need to remove comments in three lines in `PA1.Main`.

Finally, Figure 1 (right) shows the shows the output from the pretty-printer and the interpreter. We provide the MainClass example in the starter kit, but note that the starter kit program skeletons do not contain enough functionality to either pretty-print or interpret the MainClass example.

2.2 Pretty-Printer

The output must meet these requirements:

- Each statement is printed on a separate line.
- The body of an if or while statement is indented by four spaces from the first line.
- The body of an if or while statement is always surrounded by curly braces. The opening curly brace is on the same line as the if or while. The closing curly brace is on a line by itself.
- A print statement is formatted like this: `print(expr);`.
- Arithmetic expressions are fully parenthesized. For example, print `(3 + (4 * 5))`, not `3 + 4 * 5`. This will ensure your pretty-printed code matches the meaning of the AST.
- Whitespace within a line is up to you.

2.3 Test Cases

To help you find bugs in your project solutions (and thus to score more points on your assignments), we will allow you to compare your solutions against our reference solution. We won't show you the source code of the solution, of course, but we'll allow you to remotely run our compiler (or interpreter, as the case may be).

The idea is that you write test cases (that is, inputs for testing the compiler or interpreter) for your solution and submit them via CVS to your account on the instructional machines. A script will pick up these test cases from your CVS repository, run your test cases on our solution, then on your solution, and compare the two outputs. After that, we will let you know if there is a mismatch. If there is, there is a bug either in your solution or in our solution (the latter is possible but somewhat less likely).

We call this approach **remote testing**.

You will need to learn more about the remote testing infrastructure on the course web page, under Software. Remote testing is a recently developed part of the course, and so you should check the web site often for updates and bug fixes.

One important thing to remember with remote testing is that if you want to use Eclipse on your home machine, you will have to install one cs164-specific Eclipse plugin. The plugin is available on our web site, and the install process is very easy.

```
CompilationUnit
   TypeDeclaration
      SimpleName "MainClass"
      MethodDeclaration
         PrimitiveType
         SimpleName "Main"
         Block
            ExpressionStatement
               Assignment
                  SimpleName "a"
                  NumberLiteral "10"
            WhileStatement
               SimpleName "a"
               Block
                  ExpressionStatement
                     Assignment
                        SimpleName "a"
                        InfixExpression
                           SimpleName "a"
                           NumberLiteral "1"
                  ExpressionStatement
                     MethodInvocation
                        SimpleName "print"
                        SimpleName "a"
```

ASTPrinter output

```
public class MainClass {
    public static void Main() {
        a = 10;
        while(a) {
            a = (a-1);
            print(a);
        }
    }
}
```

Pretty-printer output

```
9
8
7
6
5
4
3
2
1
0
```

Interpreter output

Figure 1: Output for the AST in Figure 1

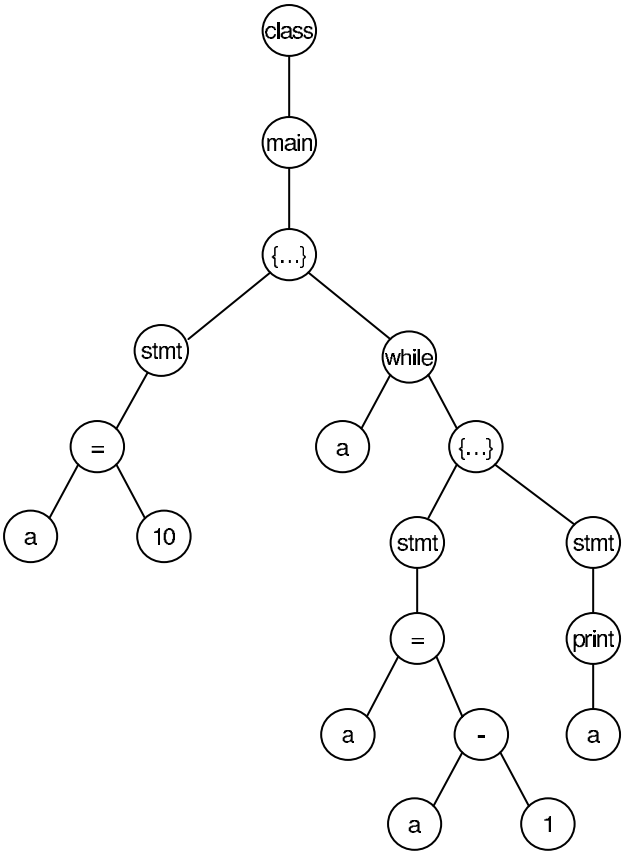

Figure 2: An example of an AST.

3 Implementation Notes

3.1 The starter kit

We've provided you with a starter kit. It contains source code for:

- A `README` file that explains how to use the kit and how to find the places you need to modify in the skeleton code. **Read this file first!**
- a skeleton of the interpreter, including sample code that handles print and integer literals. You will extend this code to implement a complete interpreter for *SkimDecaf*.
- a skeleton of the pretty-printer. You will extend this code to implement a complete pretty-printer.
- `ASTPrinter`, a class that prints out ASTs. You can use this class to view ASTs created by the Eclipse parser. Viewing ASTs for a few Java programs will help you understand the structure of the AST.
- A main method `PA1.Main()` that runs the interpreter and pretty-printer. You can modify this class as you wish, e.g., for debugging, but don't add any new functionality into it, because the TA's will overwrite it with their own file (similar to this one) when testing your assignment.
- Examples showing how to create an AST using Java code (in `TestCases.java`). These example will help you understand how the parser constructs the AST, which will in turn help you understand the AST data structure.
- The `util` package containing a wrapper for invoking the Eclipse Java parser and a class with error reporting methods that you should use instead of rolling your own. Read the files in the util package before using them. Demo the package with the `DemoHarness.java` file.

The starter kit is available on the web at

`http://www-inst.eecs.berkeley.edu/~cs164/starters/PA1.zip`.

It is also accessible on your instructional Unix account at `~cs164/public_html/starters/PA1.zip`.

The kit is designed to work with Eclipse version 3.0, which is the version of Eclipse installed on the instructional machines. Eclipse is the Java IDE that you will use for project development in this course.

You can download Eclipse 3.0 for your home machines from `www.eclipse.org`. Remember to install also the remote testing plugin from our web site.

3.2 Importing the starter kit

To import the starter kit into Eclipse:

1. Start Eclipse. Just type 'eclipse' in your Unix shell window.

2. The starter kit uses `assert`, which is a Java 1.4 feature, so you need to make sure Eclipse is configured to use Java 1.4. In the menu bar, select **Window/Preferences**. In the tree pane on the left of the Preferences dialog, select **Java/Compiler**. In the pane on the right, select Use the **Compliance and Classfiles** tab. Make sure **Compiler compliance level** is set to 1.4. Click **OK**.

3. Create a new project named **PA1**. (Go to **File/New/Project** and select Java, then click on **Next** and name the project PA1.)

4. In the menu bar select **File/Import...**.

5. Select the import source **Zip file**.

6. Under **From zip file**, select the starter kit. If you are using your instructional account, use `/home/ff/cs164/public_html/starters/PA1.zip`. If you are using another system, use the path where you saved the starter kit.

7. Make sure the project **PA1** is selected for **Into folder**.

8. Check **Overwrite resources without warning**. The starter kit overwrites the starting files for project **PA1**, so you will get some warnings if you do not check this.

9. Click **Finish**.

10. Set the command line argument for your program. This argument specifies the *SkimDecaf* program to interpret. In the menu bar, go to Run/Run.../Arguments. Set the argument to tests/sample.decaf. Later, after you implement more functionality, you will change the name to another *SkimDecaf* file in `tests`.

11. Run the starter program by opening the file `Main.java` in the editor and selecting **Run/Run As.../Java Appplication**.

3.3 Eclipse JDT

You must use the Eclipse JDT AST for this project. JDT stands for Java Development Tools. Eclipse is a generic IDE that can be configured for any programming language. JDT configures Eclipse to run as a Java IDE.

Working with the Eclipse JDT AST will give you the chance to see how the principles explained in lecture are used in a real-world programming tool. Also, this experience will help you if you ever create your own Eclipse plugins.

The JDT AST is implemented in the Java package `org.eclipse.jdt.core.dom`. You can find documentation for it in Eclipse by going to Help / Help Contents / JDT Plug-in Developer Guide / Reference / API Reference / org.eclipse.jdt.core.dom. Or simply place the cursor on the name of the class you want to learn about and press F3 to see its source code, or F4 to see its class hierarchy.

The class `ASTNode` is the top-level class in the AST node class hierarchy. Most of the other classes in this package are subclasses of `ASTNode` for specific expression and statement nodes.

The AST node classes have no public constructors. Instead, AST objects are created by calling methods on a special *factory* class `AST`. Factory is a general term for a class or method used to create objects. For example, the method `AST.newAssignment()` creates a new assignment node.

You will not need to use all the AST node types in this project. The ones you will need are:

AST Node Type	*SkimDecaf* Usage	Notes
CompilationUnit	the root of the AST	compilation unit is a different name for a source file
TypeDeclaration	class declaration	In Java, this is used for class declarations and interface declarations, but *SkimDecaf* has only a class declaration.
MethodDeclaration	method declaration	
Block	list of statements	A compound statement enclosed in curly braces. The bodies of if, while, and else are always Block AST nodes.
ExpressionStatement	expression statement	Any expression is a legal statement, called an expression statement. The expression is usually a print or assignment expression, but any expression is legal. For example, the statement `3 + 4;` is useless but legal.
IfStatement	if statement	
WhileStatement	while statement	
MethodInvocation	print expression	In *SkimDecaf* the prit expression is a method call with one argument.
Assignment	assignment expression	
InfixExpression	arithmetic expression	
SimpleName	variable name	
NumberLiteral	integer literal	

To see how these nodes are to qbe connected to form a *legal* AST, write a simple *SkimDecaf* program and run it through the Eclipse q parser and view the AST created by the parser with the `ASTPrinter` class.

3.4 The Visitor Pattern

You will use the Visitor pattern to interpret and pretty-print AST nodes. The Visitor pattern is an instance of something called a design pattern.

Design patterns were introduced to help communicate experience in designing object-oriented software. Programmers encounter similar design problems over and over again. Similar problems usually have similar solutions, although the details vary. Thus, experienced programmers can recognize a familiar problem and apply a variation of the familiar solution to that problem. Design patterns are meant to allow new programmers to benefit from this experience.

According to the classic reference *Design Patterns*, by Erich Gamma, et. al., a design pattern has four parts: a name, so we can easily talk about it; a problem statement; a generic solution to the problem; and the consequences of the applying the pattern.

We provide a brief explanation of the Visitor pattern, using the visitor provided with the Eclipse JDT AST as an example. You can find more information in the deign pattern book, as well as in the Lab Section notes (to be posted on the web site).

- **Pattern Name.** Visitor.
- **Problem.** Imagine a large, complex class hierarchy, such as a hierarchy of AST nodes. Now imagine that we need to implement several operations on the hierarchy. In our example the operations are pretty-printing and interpretation. The traditional object-oriented solution is to add a method for each operation. For example, every class would have a `prettyPrint` method and an `interp` method. Unfortunately, this scatters the code for each operation among many classes, making it hard to understand. Also, it interleaves the code for the different operations. Finally, we don't want to modify the classes directly, because they are part of the Eclipse JDT. We would have to redo our modifications every time a new version of Eclipse came out.
- **Solution.** Instead of adding methods to the class hierarchy, we will create a class for each operation. This class, called a visitor class, has a method for each type in the hierarchy. In the example, there is a method for each AST node type. The `Interpreter` class looks like this:

```
class Interpreter extends ASTVisitor {
    public boolean visit(Assignment n) {
        ...
    }
    public boolean visit(WhileStatement n) {
        ...
    }
    ...
}
```

  This class knows how to interpret every class in the AST hierarchy using the appropriate method. Also, all the visitors extend `ASTVisitor`, so they have a uniform interface.

  Now, we need only add an `accept` method to each class in the AST hierarchy. It looks like this:

```
class WhileStatement {
    ...
    public void accept(ASTVisitor n) {
        n.visit(this);
    }
    ...
}
```

  Now, we can apply any visitor to the hierarchy, and we can easily create new visitors without modifying the hierarchy.

- **Consequences.** Visitor has two main benefits: it makes it easy to add new operations, and it encapsulates each operation in a separate chunk of code. A minor benefit is that the visitor object provides a convenient place to maintain state during the operation. The main disadvantage is that it becomes harder to add new classes to the hierarchy. Adding a new class requires adding a new method to each visitor. Another disadvantage is that the classes in the hierarchy must expose enough functionality to support the visitors, breaking encapsulation. For example, in a standard object-oriented solution the Eclipse designers might have kept the list of statements in a block as private data. Since they used the Visitor pattern, they had to create a public `statements()` method to provide this information to visitors.

  In our case, the main disadvantage, that Visitor makes it harder to add new classes, does not apply. The Java language (as well as *SkimDecaf*) has a well-known specification that changes very little, so new AST classes will rarely need to be added.

The designers of Eclipse foresaw the need for AST visitors, so they provided a visitor interface, `ASTVisitor`, and `accept` methods for each node type.

There are a few details to be aware of in using the Eclipse visitors. First, in a vanilla visitor, the visit methods all return void. In Eclipse, they return a Boolean value which can be used to control traversal of the AST. If the visit method returns true, `accept` will automatically visit the children of the current node. You need finer control over the traversal order, so all of your visit methods will return false. Your visit methods will traverse the AST by calling `accept` on child nodes.

Second, ASTVisitor is an abstract class, so if you extend it directly, you will need to write a visit method for every AST node type, including many you do not need for your project. Instead, extend GenericVisitor, which implements do-nothing methods for each node type.

4 Requirements

General Requirements

All code must be in the files provided in the starter kit. You may not create new files. If you want to create a new class, create it as an inner class in one of the starter kit files.

All code must be in a Java package named `edu.berkeley.cs164.interp`. This package has been created for you in the starter kit.

Interpreter

The interpreter must be in a class called `Interpreter`, which must extend `GenericVisitor`. The interpreter must interpret *SkimDecaf* according to the language definition in this handout.

Pretty-Printer

The pretty-printer must be in a class called `PrettyPrinter`, which must extend `GenericVisitor`.

Handing in the Assignment

You will submit the assignment using CVS. Recall that you will use CVS also to upload your solution and test cases for remote testing. To submit your solution, you will check in your solution exactly as for remote testing, except that you will include in your project a file named DONE, which will tell our scripts that this is your final check in. The file named DONE should reside at the top level of your directory hierarchy.