Beginner's Guide to Using SableCC with Eclipse

Introduction

This tutorial has arisen out of the greatest dichotomy I've ever seen in regards to ease of use:
1) SableCC is one of the easiest to use compiler-compilers out there, and its design decisions were motivated by careful research.
2) SableCC is bloody hard to install if you don't know precisely what you're doing, and any documentation out there seems to assume that you're either using Linux or are intimately familiar with how Eclipse works.
Contrast this to, say, Antlr (an LL(k) parser-generator that kind of evolved into being, but has incredible amounts of useful documentation and tutorials), and you'll see what I mean. So, my aim is to level the playing field a bit. SableCC is an effective means of rapidly developing a fast compiler, and I mean to teach you how to use it fluidly in Eclipse with minimal pain.

Initial Setup

First of all, there's something you need to know about Eclipse: it defines its own "classpath" variable. So don't bother setting any environment variables unless you want to use SableCC outside of Eclipse. Rather, here's what you need to do:
1) Download Eclipse and install it.
2) Download SableCC and unzip it (I reccomend WinRAR, as it can unzip practically anything, and can compress to a greater degree than most other formats.) Extract SableCC to whatever directory you choose (mine went to **C:/sablecc-2.18.2**.)

The Testing Ground

3) Start Eclipse, and create a new project called "Test Sable Project"

a) Go to File->New->Project

b) Expand "Java" and choose "Java Project".

c) In the next window, type "Test Sable Project" for the Project name, create it wherever you want to, and set whatever project laout you desire. Click "Finish"

4) Add a new file to your project (right click on your project in the Package Explorer, choose New->File) named **simpleAdder.sable** That's right, we're making the most basic thing ever. Click "Finish"

5) What follows is a very simple Sable grammar that takes expressions of the form [INT] + [INT]; and prints the result. Copy and paste it into your new file, and save it.

```
/* simpleAdder.sable - A very simple program that recognizes two integers being added. */
Package simpleAdder ;

Helpers
    /* Our helpers */
    digit = ['0' .. '9'] ;
    sp = ' ' ;
    nl = 10 ;

Tokens
    /* Our simple token definition(s). */
    integer = digit+ sp*;
    plus = '+' sp*;
    semi = ';' nl?;

Productions
    /* Our super-simple grammar */
    program = [left]:integer plus [right]:integer semi;
```

6) Time to invoke SableCC. Remember how I told you earlier that Eclipse manages its own classpath variables? Well, these change from project to project. We'll alter the classpath now to save you some headaches later. Right-click on your project, click "Properties", and then select "Java Build Path" from the menu on the left. Select the "Libraries" tag, and click "Add External Jars...".

7) Browse to the directory you unzipped Sable to, and double-click on "lib". Then, click on the "sablecc.jar" file (the only jar in there) and click "Open". Voila! It's now in the build path! Click "OK".

Creating a SableCC Tool

8) Now that you've set the path, we'll create a tool to help you quickly compile *any* .sable file. This will save you tons of time in the future, since you only need to do it once. Select your simpleAdder.sable file, click on Run -> External Tools -> External Tools..., then click on "Program" and hit "New".
9) I called my tool "SableCC Compiler", but you can take some artistic license here. The location should be set to the location of your sdk's javaw.exe file (mine is **C:\j2sdk1.4.2_06\bin\javaw.exe**), and the working directory should be set to **${container_loc}**. Set the arguments to **-classpath C:\sablecc-2.18.2\lib\sablecc.jar org.sablecc.sablecc.SableCC ${resource_name}**, which translates as follows:

a)*-classpath* means you're setting the classpath.

b)*C:\sablecc-2.18.2\lib\sablecc.jar* is the location of the .jar file we specified earlier.

c)*org.sablecc.sablecc.SableCC* is the file that contains the Main class for invoking SableCC. It took me ages to find this, although I guess I should have checked the meta info first. Regarless, this shouldn't change for your project.

a)*${resource_name}* is the file loaded by the tool any given time it's invoked.
10) Click "Apply", then "Run". You should get a whole bunch of console text explaining what SableCC's doing. Now, hit "F5" to refresh your project's listing in the Package Explorer; there should be several new folders just recently created by SableCC.

Testing our Compiled Compiler

11) Now, to test if our grammar was added correctly, we need an Interpreter. Right-click on your project, select New->Package, and name this package simpleAdder.interpret
12) Right-click on your new package, and select New->Class. Call your class **Interpreter**, and enter the following code:

```
/* An interpreter for the simple math language we all espouse. */
package simpleAdder.interpret;

import simpleAdder.node.* ;
import simpleAdder.analysis.* ;
import java.lang.System;

public class Interpreter extends DepthFirstAdapter {

   public void caseAProgram(AProgram node) {
```

```
      String lhs = node.getLeft().getText().trim();
      String rhs = node.getRight().getText().trim();
      int result = (new Integer(lhs)).intValue() + (new Integer(rhs)).intValue();
      System.out.println(lhs + "+" + rhs + "=" + result);
   }
}
```

13) Now, we need a Main file for running the whole shebang. Right-click on your project, choose New->Class, and call it **Main**. Enter the following code:

```
/* Create an AST, then invoke our interpreter. */
import simpleAdder.interpret.Interpreter;
import simpleAdder.parser.* ;
import simpleAdder.lexer.* ;
import simpleAdder.node.* ;

import java.io.* ;

public class Main {
   public static void main(String[] args) {
      if (args.length > 0) {
         try {
            /* Form our AST */
            Lexer lexer = new Lexer (new PushbackReader(
               new FileReader(args[0]), 1024));
            Parser parser = new Parser(lexer);
            Start ast = parser.parse() ;

            /* Get our Interpreter going. */
            Interpreter interp = new Interpreter () ;
            ast.apply(interp) ;
         }
         catch (Exception e) {
            System.out.println (e) ;
         }
      } else {
```

```
            System.err.println("usage: java simpleAdder inputFile");
            System.exit(1);
        }
    }
}
```

14) Time for you to run this monster. Right-click on your Main.java file and select Run->Run... Click "New", and a configuration should be created automatically for your project. Tab over to "Arguments", and enter "tester.sa" in the "Program Arguments" box. (You'll need to download this file here, or supply a similar test file. Alternatively, you could map the user's input.) Click "Run" and watch your program execute. Ta-da!

More Stuff

15) Anytime you change your grammar (in our case, simpleAdder.sable), you'll need to run your SableCC tool on it to re-generate the files for your parser/lexer/etc.
16) If you're just changing the interpreter, you merely need to recompile the java code. This is one of the great strengths of SableCC.
17) It is a Very Good Idea to clean (read: "delete") any files SableCC has generated before re-generating your grammar. Otherwise, you might get old (and probably incorrect) code conflicting with new (correct) code.

Epilogue

Special thanks to Alan Oursland's online tutorial for setting up Antlr. Midway through reading his guide, I realized exactly what I was doing wrong with SableCC.
Thanks also is extended to James Handley, for explaining how Sable's walker class works. Sample code on this page is an (extremely) dumbed-down version of that provided for his class.
Also, I take no responsibility for the computational brevity (or, for that matter, accuracy) of this tutorial. There are probably better ways of invoking Sable then I've detailed here (Apache Ant seems a likely alternative.) Moreover, the actual example used here is morbidly simple, and not a good example of what Sable's really good for. However, in the interest of helping out struggling coders, I'm afraid that the blind will have the lead the blind for a while.

Got comments or suggestions relating to this guide? Feel free to Mail Me your thoughts.