Comp 411 Putative Assignment

Putative Assignment: Symbolic Evaluation of Boolean Expressions in Java

Background

Comp 411 presumes familiarity with functional programming in Scheme and mastery of object-oriented design in Java, most notably how to write functional programs (only involving immutable objects) in Java. Historically, the courses Comp 210 and 212 (and later Comp 211) covered this material. Several years ago, the core programming curriculum was revised, de-emphasizing functional programming and object-oriented design. As a result, it is unclear how well undergraduate students are prepared for this course. This web page presents a functional programming assignment in Java where the behavior of the Java is specified by a purely functional program in Scheme. Students who enroll in this course should be comfortable tackling this assignment (taken directly from Comp 211 in Spring 2011).

Overview

Write a Java program `BoolSimp.java` that reduces boolean expressions (represented in the input and output streams in Scheme-like notation) to simplified form. For the purposes of this assignment, boolean expressions are Scheme expressions constructed from:

- the symbols `T` and `F` denoting the boolean values `true` and `false` ;
- boolean variables (represented by symbols other than `T`, `F`, `!`, `&`, `|`, `>`, and `?` that can be bound to either `true` or `false` .
- the unary function `!` meaning `not` .
- the binary functions `&`, `|`, and `>` denoting `and`, `or`, and `implies`, respectively), and
- the ternary function `?` meaning `if` .

The shorter names `T`, `F`, `!`, `&`, `|`, `>`, and `?` are used instead of `true`, `false`, `not`, `and`, `or`, `implies`, and `if` for notational brevity which matters in very large inputs.

Some sample inputs are:

- T
- F
- x
- (! x)
- (| x (! x))
- (& x (! x))
- (> x x)

These formulas are represented internally using abstract syntax (implemented using the composite pattern) as defined in the Java stub file BoolSimp.java. Only the last four formulas can be simplified; the preceding formulas reduce to themselves. Some much more complex sample inputs can be found in the file littleData1.

The support code for this assignment includes:

- a Scheme program in the file boolsimp.ss equivalent to the Java program that you are required to write;
- a Java "stub" file `BoolSimp.java` for your program that defines a composite hierarchy of "abstract syntax" tree classes rooted in the class `Form` representing boolean expressions;
- a Java library file `Parser.java` containing a class `Parser` with
  - a `read()` method that reads a boolean expression represented in "Scheme form" and returns the corresponsing Java `Form` abstract syntax tree and
  - a `reduce()` method that composes the visitors you must write in to reduce whatever formula the `Parser` instance contains to simplified form.
- a Java "stub" test file `BoolSimpTest.java` that includes some rudimentary tests of the code in the `BoolSimp.java` stub file.

The stub file `BoolSimp.java` also includes comments showing you exactly what code you have to write to complete writing your simplifier. Of course, you also need to write corresponding tests and add them to the file `BoolSimpTest.java`.

The file `Parser.java` is provided to enable you to test your solution on large inputs stored in files. `Parser.java` includes two `Parser` constructors `Parser(File file)` and `Parser(String form)` for building parsers to parse the boolean expression (in external text form) in the specified `File` or `String`, respectively. Since the library class `File` is defined in the package `java.io`, you need to insert

```
import java.io.File;
```

at the head of a test file that uses the `Parser` class on the contents of a file.
To construct a `Parser` for the formula in a file `<fileName>` you must invoke

```
new Parser(new File("<fileName>"));
```

If you omit the `new File(...)` construction in the argument to `Parser` and use `"<fileName>"` instead, you will create a `Parser` for the String `"<file Name>"`. which is interpreted as a simple boolean variable. The `File` input format is important because it enables us to conveniently apply your simplifier to formulas that are thousands of symbols long. As a result, you only have to translate the Scheme code in `boolsimp.ss` into corresponding cleanly-written OO Java code by filling in the gaps in our Java stub file `boolSimp.java`. You are expected to appropriately use the composite, interpreter, singleton, and visitor patterns in the code that you write. Since the only stub files that you have to modify are `boolSimp.ava` and `boolSimpTest.java`, your assignment is to create expanded versions of these files including a comprehensive test suite in boolSimpTest.java. *Warning:* your program must handle large inputs like large test files provided below.

The Scheme file `boolsimp.ss` includes Scheme functions `parse` and `unparse` to translate Scheme lists into abstract syntax trees and vice-versa. Scheme provides a simple external syntax for lists (consonant with its LISP heritage) but Java does not. Hence the Java `Parser` class works on Java strings instead of lists. The Java visitor class `Print` in the `BoolSimp.java` file performs unparsing of the abstract syntax types `Form` and `IfForm` to type String.

The Scheme parsing functions rely on the following Scheme data definitions.

Given

```
(define-struct ! (arg))bigData0
(define-struct & (left right))
(define-struct \| (left right))
(define-struct > (left right))
(define-struct ? (test conseq alt))
```

a `boolExp` is either:

- a boolean constant `true` and `false` ;
- a symbol `S` representing a boolean variable;
- `(make-Not X)` where `X` is a `boolExp` ;
- `(make-And X Y) where X` and `Y` are `boolExps`;
- `(make-Or X Y) where X` and `Y` are `boolExps`;
- `(make-Implies X Y) where X` and `Y` are `boolExps`; or
- `(make-If X Y Z)` where `X`, `Y`, and `Z` are `boolExps`.

Note: The `or` operator must be written as

```
\|
```

in Scheme instead of `|` because `|` is a metasymbol with a special meaning in Scheme.

Description of the Provided Scheme program

Given a parsed input of type `boolExp`, the simplification process consists of following four phases:

- Conversion to `if` form implemented by the function `convert-to-if` .
- Normalization implemented by the function `normalize` .
- Symbolic evaluation implemented by the function `eval` .
- Conversion back to conventional *boolean* form implemented by the function `convert-to-bool` .

These phases are described in detail in HW6 from Comp 211.

Hints on Writing Your Java Code

The Java abstract syntax classes include a separate composite hierarchy (called `IfForm`) for representing boolean expression as conditionals (the type `if Exp` in `boolsimp.ss`). This representation includes only three concrete variant classes, making it much easier to write the visitors that perform normalization, evaluation, and clean-up.

The visitor pattern is a straightforward but notationally involved alternative to the interpreter pattern. If you do not have much experience writing and debugging Java code involving visitors, we suggest that you write a solution using the interpreter pattern first and then translate your interpreter pattern code to visitor pattern code. (Perhaps IDEs like Eclipse should support such transformations.)

Support Code

Here are the links for the files:

- boolsimp.ss is the reference Scheme program.
- BoolSimp.java is a stub program for a visitor solution.
- BoolSimpTest.java is a stub test file for a visitor solution.

- Parser.java is a parser file for a visitor solution.

Sample Input Files

The following files contain large formulas that can be reduced by your simplifier. Only the files named bigData x may require a larger thread stack size than the JVM default on most platforms. NOTE: to handle the files bigData0 and bigData1, you may need to pass the JVM argument -Xss64M for the Interactions JVM using the DrJava Preferences command on the Edit menu. The JVM argument setting can be found on the last panel (called JVMs) in the Preferences categories tree.

- `littleData1 -> "T"`
- `littleData2 -> "T"`
- `littleData3 -> "(> h (> g (> f (> e (> d (> c (! b)))))))"`
- `littleData4 -> "(> h (> g (> f (> e (| d (| c (| b a)))))))"`
- `bigData0 -> "T"`
- `bigData1 -> "(> j (> i (> h (> g (> f (> e (| d (| c (| b a)))))))))"`