

HW08

Homework 08: Symbolic Evaluation of Boolean Expressions in Java

Due: 11:59pm Monday, Nov 16, 2020

200 points

Overview

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

Note that the concrete (surface) syntax for boolean expressions differs from Assignment 5 because we are using 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. You can write your Java code to perform boolean simplification using either the interpreter pattern or the visitor pattern. The latter is perhaps slightly harder since there is more notational overhead, but it is valuable practice if you anticipate writing significant Java applications involving immutable inductively defined data

The course staff is providing

- the Java "stub" files `boolSimp.java` (where the functions in the simplifier are written using the visitor pattern) and `interpBoolSimp.java` (where the functions in the simplifier are written using that defines a composite hierarchy of "abstract syntax" tree classes rooted in the class `Form` representing boolean expressions);
- a Java library file `Parser.java` contain a class `Parser` with
 - a `read()` method that reads a boolean expression represented in concrete syntax ("Racket form") and returns the corresponding Java `Form` abstract syntax tree; and
 - a `reduce()` method that composes the visitors you must write in `boolSimp.dj` 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` (and similarly `interpBoolSimp.java`) stub file.

The stub files `boolSimp.java` and `interpBoolSimp.java` also include 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 primarily 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 concrete syntax form) in the specified `File` or `String`, respectively. Since the Java library class `File` is defined in the package `java.io`, which is not imported by default (unlike `java.lang`), you need to insert either

```
import java.io.*;
```

or more specifically

```
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 can invoke

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

If you omit the `new File(...)` wrapper around the `String` name of the file and simply use `"<fileName>"` instead, you will create a `Parser` for the `String "<fileName>"`, which is then interpreted as a simple boolean variable. The `File` input medium is important because it enables us to conveniently apply your simplifier to formulas that are thousands of symbols long. As a result, for this assignment you only have to translate the Racket code in `boolSimp.p.rkt` into corresponding cleanly-written OO Java code by filling in the gaps in our Java stub file `boolSimp.java` (or `interpBoolSimp.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.java` (or `interpBoolSimp.java`) and `boolSimpTest.java`, simply upload working versions of these files to the Rice SVN repository to "turn in" your assignment. *Warning:* we will run your program on large inputs to determine if you wrote the code correctly. Try using the large test files provided on the course wiki.

All of the support files for this assignment are written `standard` Java rather than functional Java because the functional Java translator embedded no longer works well enough to write non-trivial programs like a boolean simplifier. In principle, a very nice solution to this problem can be written in functional Java. If I (with the help of students in Comp 402/501) manage to rewrite/repair the functional Java translator this spring, future editions of this assignment may be conducted entirely in functional Java. We expect you to write functional code (mutation of data structures or the cells holding the bindings of variables). If you use DrJava as your IDE, make sure that the language is set to "full Java".

The Racket file `boolSimp.rkt` includes Racket functions `parse` and `unparse` to translate Racket lists into abstract syntax trees and vice-versa. Racket provides a simple external syntax for lists (in homage to 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`.

As in Homework 5, the Racket parsing functions in `boolSimp.rkt` rely on the following Racket data definitions.

Given

```
(define-struct Not (arg))
(define-struct And (left right))
(define-struct Or (left right))
(define-struct Implies (left right))
(define-struct If (test consequent))
```

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`.

Notes:

1. The `or` operator must be written as `|` in Racket instead of `|` because `|` is a metasymbol with a special meaning in Racket.
2. In essence, `boolSimp.rkt` is a solution to Homework 5. The Java code in the file `Parser.java` assumes the input is written in Racket notation, but with the following abbreviations to shorten the length of formulas:

Abbreviation	Original Symbol
T	true
F	false
!	Not
&	And
	Or
>	Implies
?	If

3. The Java abstract syntax classes include a separate composite hierarchy (called `ifExp` in `boolSimp.rkt`). This representation includes only three concrete variant classes, making it much easier to write the visitors that perform normalization, evaluation, and convert-To-Bool.

Hints on Writing Your Java Code

The visitor pattern is a straightforward but notationally cumbersome alternative to the interpreter pattern. You can mechanically translate interpreter pattern code to visitor pattern code. (Perhaps IDEs like Eclipse should support such transformations.) The interpreter solution to this assignment is a bit easier to write than the visitor solution. If you are still learning Java mechanics, you are encouraged to write an interpreter solution first and perhaps translate it later to visitor form. A perfect visitor solution will be given 10 extra points over a perfect interpreter solution. If you submit an interpreter solution, your program must conform to class signatures given in the interpreter pattern support code below (just as a visitor solution must conform to the class signatures given in the visitor pattern code below).

The interpreter version of the support code replaces the `ConvertToIf`, `Normalize`, `HeadNormalize`, `Evaluate`, and `Print` visitors by methods named `convertToIf`, `normalize`, `headNormalize`, `eval`, and `print`. The classes `Parser.java` and `InterpParser.java` contain references to these visitor class names and method names, respectively.

Support Code

Here are the links for the files:

- [boolSimp.rkt](#) is the reference Racket 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.
- [interpBoolSimp.dj](#) is a stub program for an interpreter solution.
- [interpBoolSimpTest.java](#) is a stub test file for an interpreter solution.
- [InterpParser.java](#) is a parser file for an interpreter solution.

`InterpParser.java` is distinct from `Parser.java` because the code for the `reduce` method embedded in the parser is different in the two versions.

Sample Input Files

The following files contain large formulas that can be reduced by your simplifier. Only the files named `bigData x` require a larger thread stack size than the JVM default on most platforms. **NOTE:** to handle the `bigData x` files, you must set 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))))))"