Computer Science C24
University of Toronto Scarborough

Homework Exercise # 2

Handing in and marking

For all exercises/assignments in this course, you need to submit your solutions to the pencil-and-paper questions on crowdmark and your solutions to the programming questions on MarkUs. Your pencil-and-paper solutions will be marked with respect to correctness, clarity, brevity, and readability. Your code will be marked with respect to correctness, efficiency, program design and coding style, clarity, and readability. This exercise counts for 10% of the course grade.

Question 1. Functional Programming in Racket

Recall the context free grammar for boolean expressions from your first exercise (with a slight modification for the primitives).

```
<boolean-expr> ::= <disjunction>
<disjunction> ::= <disjunction> or <conjunction> | <conjunction>
<conjunction> ::= <conjunction> and <negation> | <negation>
<negation> ::= not <negation> | <bracket>
<bracket> ::= (<boolean-expr>) | <primitive>
<primitive> ::= #t | #f | <symbol>
<symbol> ::= ...
```

For this question we would like to write an evaluator for expressions generated by this grammar extended with implication, in Racket. As we quickly realize that writing a real parser for this language is not quite trivial, we make our lives much easier by insisting that all implicit parentheses are always specified.

The input to the evaluator will be in a form of a Racket list, or a single truth value, or a single variable. For example, `(x and y or not z)` would be a valid expression in our original grammar, but for our evaluator it has to become `((x and y) or (not z))`.

The context, in which an expression can be evaluated is an assignment of values to varibles in the expression. We will represent such an assignment by a list of Racket pairs. For example, `((a . #t) (b . #f) (c . #f) (d . #t))` means that `a` has value true, `b` has value false, etc.

1. Your first task is to implement the evaluator for boolean expressions in this language.
   - Carefully study the starter code and documentation provided in the module `evaluator` before you complete the implementation.
   - Think carefully about designing your solution. Clear, easy to read, modular code will receive many more marks. How easy is it to add another operator (for example, `iff`) to your solution?
2. Sometimes, even if we do not have values of all the variables in a boolean expression, we can still determine its value. For example, the value of `((x and y) or (not z))` in context `((z . #f))` is true, even though we do not know the values of `x` and `y`.

   Often, even if we cannot determine the value of an expression, we can greatly simplify it. For example, the expression `((x and y) or (not z))` in context `((z . #t))` can be simplified to `(x and y)`.

   Your second task is to implement the simplifier. To make the task easier, you only need to apply the following simplification rules to the expressions:
   - `(#t and x) == x` and `(x and #t) == x`
   - `(#f and x) == #f` and `(x and #f) == #f`
   - `(#t or x) == #t` and `(x or #t) == #t`
   - `(#f or x) == x` and `(x or #f) == x`
   - `(#t implies x) == x` and `(x implies #t) == #t`

- `(#f implies x) == #t` and `(x implies #f) == (not x)`
- `(not (not x)) == x`

Of course, you still need to evaluate as much as possible of the given expression under the given context, and simplify the expression as much as you can, given the above rules. For example, the expression `(not (a implies b))` with context `(b . #f)` should simplify to `a`.

Once again,

- Carefully study the starter code and documentation provided in the module `evaluator` before you complete the implementation.
- Think carefully about designing your solution. Clear, easy to read, modular code will receive many more marks. How easy is it to add another operator (for example, `iff`) to your solution?

Question 2. Exploring Programming Languages: Functional Features of Modern Java

In this question we will implement two versions of the equivalent of the Racket module `evaluator` from the previous question, in Java. The first version will be purely **Object-Oriented** ("traditional" Java) and the second version will use **functional features** of Java (first introduced with version 8).

Carefully study the UML diagrams and the starter code, and implement all the required classes accordingly. In addition, implement the methods in the file `BooleanExpressions.java`. Make sure to follow the instructions in the starter code: **no loops! Use Java's `Streams` and methods `map` and `reduce`**.

Notice that the method `evaluate` throws an `UnassignedVariableException`, if there is at least one variable in the expression that is not assigned a value in the argument `context` (even if it is technically possible to evaluate the expression using simplification rules).

For the second version, make sure to study the starter of the file `Negation.java`: it shows how to use the **Java lambda expressions and method references**.

In this exercise, we will implement more simplification rules than we did in the first question:

- `(#t and x) == x` , `(x and #t) == x`
- `(#f and x) == #f` , `(x and #f) == #f`
- `(#t or x) == #t` , `(x or #t) == #t`
- `(#f or x) == x` , `(x or #f) == x`
- `(#t ==> x) == x` , `(#f ==> x) == #t`
- `(x ==> #t) == #t` , `(x ==> #f) == (not x)`
- `(#t <==> x) == x` , `(#f <==> x) == (not x)`
- `(x <==> #t) == x` , `(x <==> #f) == (not x)`
- `(not (not x)) == x`
- `(x and x) == x` , `(x or x) == x`
- `(x ==> x) == #t`
- `(x <==> x) == #t`

Make sure all your classes/interfaces are **public**, all methods/fields have access modifiers exactly as in the diagram, all classes/interfaces and methods are named **exactly as specified**, and are in the **package booleanoo** and **package booleanoofunc**.

Make sure your code passes the `checkstyle` tool, linked from the course website. As always, **make sure you are using the correct versions of all tools**.

As always, make sure to fully test your code. The starter code provides **example** usage of the code you will develop, but **not proper testing** of the code — it is your responsibility to thoroughly test all code you submit.

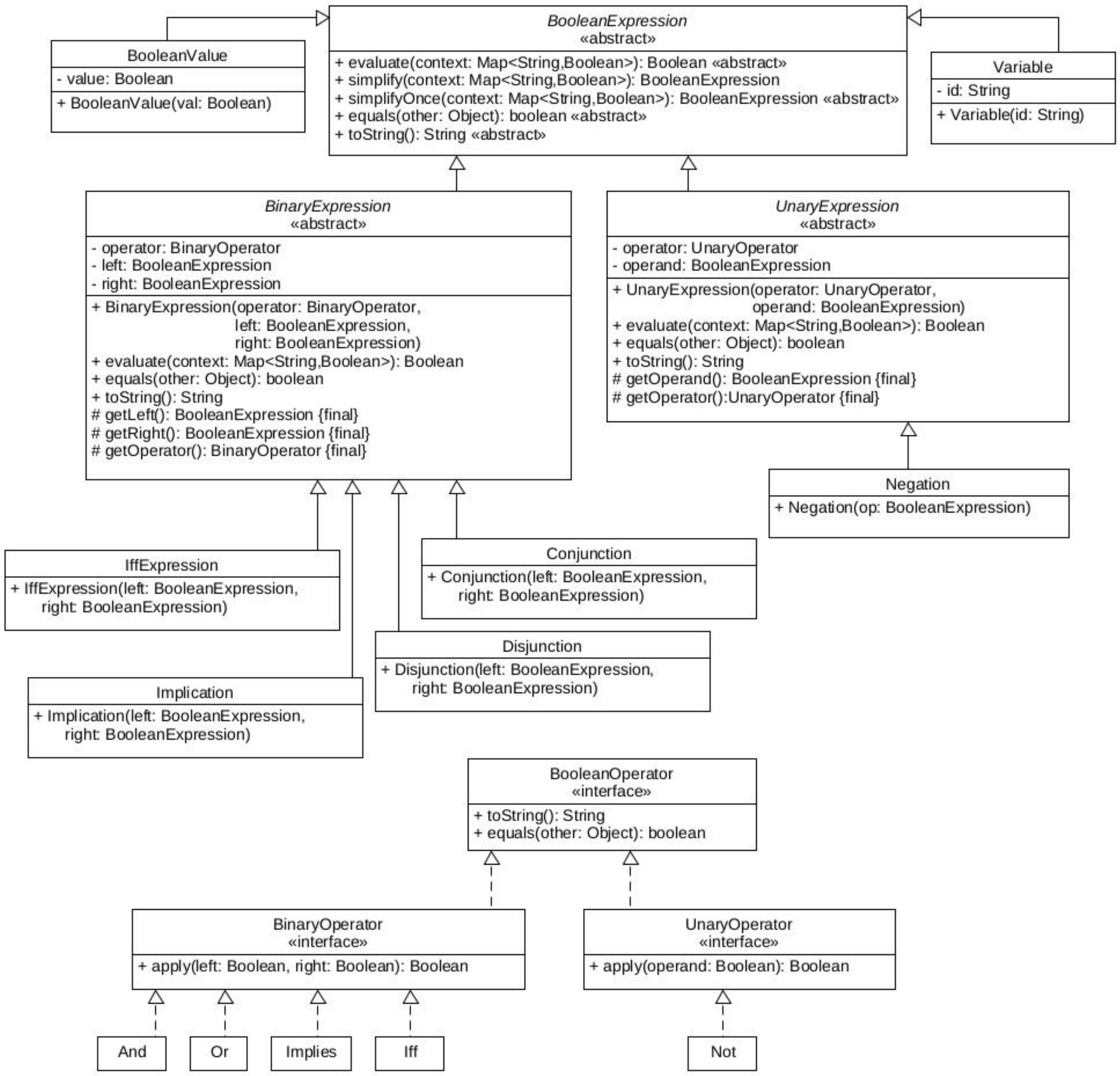
BooleanExpression
«abstract»
+ evaluate(context: Map<String,Boolean>): Boolean «abstract»
+ simplify(context: Map<String,Boolean>): BooleanExpression
+ simplifyOnce(context: Map<String,Boolean>): BooleanExpression «abstract»
+ equals(other: Object): boolean «abstract»
+ toString(): String «abstract»
BooleanValue
- value: Boolean
+ BooleanValue(val: Boolean)
Variable
- id: String
+ Variable(id: String)
BinaryExpression
«abstract»
- operator: BinaryOperator
- left: BooleanExpression
- right: BooleanExpression
+ BinaryExpression(operator: BinaryOperator, left: BooleanExpression, right: BooleanExpression)
+ evaluate(context: Map<String,Boolean>): Boolean
+ equals(other: Object): boolean
+ toString(): String
# getLeft(): BooleanExpression {final}
# getRight(): BooleanExpression {final}
# getOperator(): BinaryOperator {final}
UnaryExpression
«abstract»
- operator: UnaryOperator
- operand: BooleanExpression
+ UnaryExpression(operator: UnaryOperator, operand: BooleanExpression)
+ evaluate(context: Map<String,Boolean>): Boolean
+ equals(other: Object): boolean
+ toString(): String
# getOperand(): BooleanExpression {final}
# getOperator():UnaryOperator {final}
Negation
+ Negation(op: BooleanExpression)
IffExpression
+ IffExpression(left: BooleanExpression, right: BooleanExpression)
Conjunction
+ Conjunction(left: BooleanExpression, right: BooleanExpression)
Disjunction
+ Disjunction(left: BooleanExpression, right: BooleanExpression)
Implication
+ Implication(left: BooleanExpression, right: BooleanExpression)
BooleanOperator
«interface»
+ toString(): String
+ equals(other: Object): boolean
BinaryOperator
«interface»
+ apply(left: Boolean, right: Boolean): Boolean
UnaryOperator
«interface»
+ apply(operand: Boolean): Boolean
And
Or
Implies
Iff
Not

Figure 1: Object-Oriented “traditional”

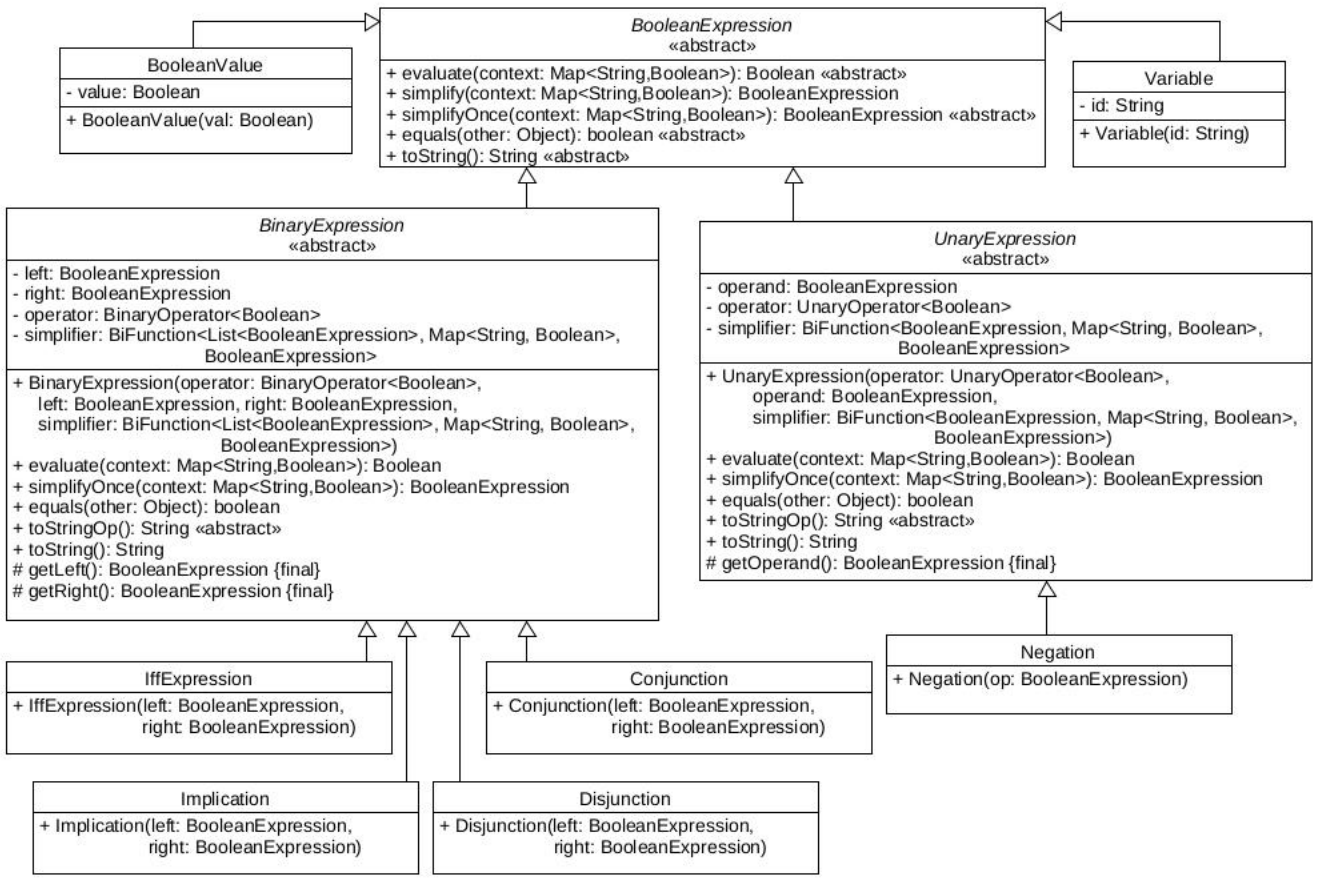
BooleanExpression
«abstract»
+ evaluate(context: Map<String,Boolean>): Boolean «abstract»
+ simplify(context: Map<String,Boolean>): BooleanExpression
+ simplifyOnce(context: Map<String,Boolean>): BooleanExpression «abstract»
+ equals(other: Object): boolean «abstract»
+ toString(): String «abstract»
BooleanValue
- value: Boolean
+ BooleanValue(val: Boolean)
Variable
- id: String
+ Variable(id: String)
BinaryExpression
«abstract»
- left: BooleanExpression
- right: BooleanExpression
- operator: BinaryOperator<Boolean>
- simplifier: BiFunction<List<BooleanExpression>, Map<String, Boolean>, BooleanExpression>
+ BinaryExpression(operator: BinaryOperator<Boolean>, left: BooleanExpression, right: BooleanExpression, simplifier: BiFunction<List<BooleanExpression>, Map<String, Boolean>, BooleanExpression>)
+ evaluate(context: Map<String,Boolean>): Boolean
+ simplifyOnce(context: Map<String,Boolean>): BooleanExpression
+ equals(other: Object): boolean
+ toStringOp(): String «abstract»
+ toString(): String
# getLeft(): BooleanExpression {final}
# getRight(): BooleanExpression {final}
UnaryExpression
«abstract»
- operand: BooleanExpression
- operator: UnaryOperator<Boolean>
- simplifier: BiFunction<BooleanExpression, Map<String, Boolean>, BooleanExpression>
+ UnaryExpression(operator: UnaryOperator<Boolean>, operand: BooleanExpression, simplifier: BiFunction<BooleanExpression, Map<String, Boolean>, BooleanExpression>)
+ evaluate(context: Map<String,Boolean>): Boolean
+ simplifyOnce(context: Map<String,Boolean>): BooleanExpression
+ equals(other: Object): boolean
+ toStringOp(): String «abstract»
+ toString(): String
# getOperand(): BooleanExpression {final}
IffExpression
+ IffExpression(left: BooleanExpression, right: BooleanExpression)
Conjunction
+ Conjunction(left: BooleanExpression, right: BooleanExpression)
Negation
+ Negation(op: BooleanExpression)
Implication
+ Implication(left: BooleanExpression, right: BooleanExpression)
Disjunction
+ Disjunction(left: BooleanExpression, right: BooleanExpression)

Figure 2: Object-Oriented plus Functional