#lang racket

#|

CS 550 - Programming Languages

Drexel University Spring 2016-2017

Homework Assignment 1

Due Wed April 19 at 11:59pm

Name: Greg Matthews and Ke Yang

For this homework, you will need to use DrRacket. You should have installed

it by now on your machine, and know the basics of using it. Make sure you

have the latest version.

Instructions for using this file:

- open this file in DrRacket as assign1.rkt

- insert your solutions into this file where indicated (for instance

as "'replace-this-with-your-implementation")

- make sure the entire file is accepted by DrRacket. If you don't finish some

problems, comment them out. The same is true for any English text that you

may add. This file already contains many comments, so you can see what the

syntax is.

- Submit your homework through Blackboard (learning.drexel.edu)

- All function names are given, do not change the names of any functions

Finally, add an ASCII text file "comments.txt" into the same directory

where you put assign1.rkt, with the following contents:

(i) a summary of how much time you spent on this homework

(ii) any feedback on the homework you may have, such as errors in it,

sources of misunderstanding, general difficulty, did you learn

something, etc.

(iii) any problems that were not completed

|#

;; We use rackunit package to do unit tests. When you start,

;; all the tests will be failing. Once you implement the required

;; functions, the unit tests associated with those functions should

;; pass.

;;

;;

(require rackunit)

(require rackunit/text-ui)

#|

Part 0 : bool-eval

Below we provide a function to evaluate boolean expressions.

This evaluator supports constants, variables, not, and, or

Review this code as a starting point.

|#

;Lookup

;Lookup the value of a variable name in an evironment

;target - variable name to find

;env - A list of bindings (name value) to search.

;Error is variable name is not defined.

(define (lookup target env)

(if

(null? env)

(error 'lookup "Variable Name Not Found")

(if

(equal? (car (car env)) target)

(car (cdr (car env)))

(lookup target (cdr env))

)

)

)

;is-reserved-word

;Determine if the expression is a constant reserved word

;expression - the value to check

(define (is-reserved-word? expression)

(cond

[ (equal? expression '#t) #t]

[ (equal? expression '#f) #t]

[ (equal? expression 'or) #t]

[ (equal? expression 'and) #t]

[ (equal? expression 'not) #t]

[ (equal? expression 'implies) #t]

[ (equal? expression 'iff) #t]

;Otherwise

[ else #f]

)

)

;is-constant

;Determine if the expression is a constant

;The only two constants are #t and #f

;expression - the value to check

(define (is-constant? expression)

(or (equal? expression #t) (equal? expression #f))

)

;is-variable

;Determine if the expression is a variable

;There are only two kinds of symbols, variables or reserved words.

;If the expression is a symbol then it must be either a variable or reserved word.

;expression - the value to check

(define (is-variable? expression)

(and

(symbol? expression)

(not (is-reserved-word? expression))

)

)

;is-not

;Determine if the expression is the application of a NOT function

;The expression will be a list where the first element is the word not.

;expression - the value to check

(define (is-not? expression)

(equal? (first expression) 'not)

)

;is-or

;Determine if the expression is the application of an OR function

;The expression will be a list where the first element is the word or.

;expression - the value to check

(define (is-or? expression)

(equal? (first expression) 'or)

)

;is-and

;Determine if the expression is the application of an AND function

;The expression will be a list where the first element is the word and.

;expression - the value to check

(define (is-and? expression)

(equal? (first expression) 'and)

)

;op1

;Get the first input to a function call.

;A two input function look like (and a b)

;op1 would return a in this case.

;expression - the expression to use

(define (op1 expression)

(second expression))

;op2

;Get the second input to a function call.

;A two input function look like (and a b)

;op2 would return b in this case.

;expression - the expression to use

(define (op2 expression)

(third expression))

;bool-eval

;Given a boolean expression and list of variable bindings

;determines the value of the expression

;expr - expression to evaluate

;env - List of bindings (name value) to use for the variables.

(define (bool-eval expr env)

(cond

[(is-constant? expr) expr ]

[(is-variable? expr) (lookup expr env)]

[(is-not? expr) (not (bool-eval (op1 expr) env))]

[(is-or? expr) (or (bool-eval (op1 expr) env)

(bool-eval (op2 expr) env))]

[(is-and? expr) (and (bool-eval (op1 expr) env)

(bool-eval (op2 expr) env))]

)

)

;Tests for bool-eval

;Test Constants

(define-test-suite bool-eval-suite

(check-equal?

(bool-eval '#t '( (a #t) (b #f))) ;True

#t)

(check-equal?

(bool-eval '#f '( (a #t) (b #f))) ;False

#f)

;Test Variables

(check-equal?

(bool-eval 'a '( (a #t) (b #f))) ;True

#t)

(check-equal?

(bool-eval 'b '( (a #t) (b #f))) ;False

#f)

;Test Or

(check-equal?

(bool-eval '(or a a) '( (a #t) (b #f))) ;True

#t)

(check-equal?

(bool-eval '(or a b) '( (a #t) (b #f))) ;True

#t)

(check-equal?

(bool-eval '(or b a) '( (a #t) (b #f))) ;True

#t)

(check-equal?

(bool-eval '(or b b) '( (a #t) (b #f))) ;False

#f)

;Test not

(check-equal?

(bool-eval '(not a) '( (a #t) (b #f))) ;False

#f)

(check-equal?

(bool-eval '(not b) '( (a #t) (b #f))) ;True

#t)

;Test and

(check-equal?

(bool-eval '(and a a) '( (a #t) (b #f))) ;True

#t)

(check-equal?

(bool-eval '(and a b) '( (a #t) (b #f))) ;False

#f)

(check-equal?

(bool-eval '(and b a) '( (a #t) (b #f))) ;False

#f)

(check-equal?

(bool-eval '(and b b) '( (a #t) (b #f))) ;False

#f)

)

(run-tests bool-eval-suite 'verbose)

#|

Part 1

(bool-simp expr) returns a simplified boolean expression.

It performs the following simplifications.

1) evaluate all constant subexpressions

2) (and #t expr) -> expr

3) (and expr #t) -> expr

4) (and #f expr) -> #f

5) (and expr #f) -> #f

6) (or #t expr) -> #t

7) (or expr #t) -> #t

8) (or #f expr) -> expr

9) (or expr #f) -> expr

10) (not (not expr)) -> expr

Simplification (1) and (10) are done through the helper routine not-simp.

Simplifications (1) and (2)-(5) are done through the helper routine and-simp.

Simplifications (1) and (6)-(9) are done through the helper routine or-simp.

bool-simp traverses the boolean expression and recursively simplifies

all operands to NOT, OR and AND, then it calls the appropriate helper routine

to perform operator specific simplifiations and constant evaluation.

You should implement bool-simp, not-simp, or-simp, and and-simp.

|#

;not-simp

;Simplify the input to a NOT function.

;You can assume that the expr has already been simplified.

;You will do that in bool-simp.

;

;The input is the operand to not.

;If the user called (bool-simp '(not #t))

;then the call to not-simp will be (not-simp #t)

;

;Examples

;(not-simp #t ) returns #f

;(not-simp #f ) returns #t

;(not-simp 'a) returns (not a)

;(not-simp '(and a b)) returns (not (and a b))

;Extra Credit: If you want to do the extra credit then additional examples are

;(not-simp '(not a)) returns a

;(not-simp '(not (and a b)) returns (and a b)

(define (not-simp expr)

(cond

[(equal? expr #t) #f]

[(equal? expr #f) #t]

[(and {list? expr} (is-not? expr)) (last expr)]

[(list 'not expr)]))

;Checks

(define-test-suite not-simp-suite

(check-equal? (not-simp #t) #f)

(check-equal? (not-simp #f) #t)

(check-equal? (not-simp 'a) '(not a))

(check-equal? (not-simp '(and a b)) '(not (and a b)))

;Extra Credit 1

;If you are doing this extra credit, uncomment the below tests

(check-equal? (not-simp '(not a)) 'a)

(check-equal? (not-simp '(not (and b c))) '(and b c))

)

(run-tests not-simp-suite 'verbose)

;and-simp

;Simplify input to an AND function.

;You can assume that the expr has already been simplified.

;You will do this in bool-simp

;

;The inputs are the two operands to the AND call.

;If the user called (bool-simp '(and #t #f))

;then the call to and-simp will be (and-simp #t #f)

;

;Examples

;Constant Expressions

;(and-simp #t #t) returns #t

;(and-simp #t #f) returns #f

;(and-simp #f #t) returns #f

;(and-simp #f #f) returns #f

;Rules 2-5

;(and-simp #t '(and f g)) returns (and f g)

;(and-simp '(or h k) #t) returns (or h k)

;(and-simp #f '(not a)) returns #f

;(and-simp '(or a b) #f) returns #f

;(and-simp '(and a b) '(or c d)) returns (and (and a b) (or c d))

;(and-simp 'a 'b) returns (and a b)

(define (and-simp expr1 expr2)

(cond

[(and (equal? expr1 #t) (equal? expr2 #t)) #t]

[(or (equal? expr1 #f) (equal? expr2 #f)) #f]

[(equal? expr1 #t) expr2]

[(equal? expr2 #t) expr1]

[(list 'and expr1 expr2)]

))

;Checks

(define-test-suite and-simp-suite

(check-equal? (and-simp #t #t) #t)

(check-equal? (and-simp #t #f) #f)

(check-equal? (and-simp #f #t) #f)

(check-equal? (and-simp #f #f) #f)

(check-equal? (and-simp #t '(and f g)) '(and f g))

(check-equal? (and-simp '(or h k) #t) '(or h k))

(check-equal? (and-simp #f '(not a)) #f)

(check-equal? (and-simp '(or a b) #f) #f)

(check-equal? (and-simp '(and a b) '(or c d)) '(and (and a b) (or c d)))

(check-equal? (and-simp 'a 'b) '(and a b))

)

(run-tests and-simp-suite 'verbose)

;or-simp

;Simplify input to an OR function.

;You can assume that the expr has already been simplified.

;You will do this in bool-simp

;

;The inputs are the two operands to the OR call.

;If the user called (bool-simp '(or #t #f))

;then the call to or-simp will be (or-simp #t #f)

;

;Examples

;Constants

;(or-simp #t #t) returns #t

;(or-simp #t #f) returns #t

;(or-simp #f #t) returns #t

;(or-simp #f #f) returns #f

;Simplifications 6-9

;(or-simp #t '(and b g)) returns #t

;(or-simp '(not b) #t) returns #t

;(or-simp #f '(or a b)) returns (or a b)

;(or-simp '(and x y) #f) returns (and x y)

;(or-simp '(not a) '(not b)) returns (or (not a) (not b))

;(or-simp 'a 'b) returns (or a b)

(define (or-simp expr1 expr2)

(cond

[(and (equal? expr1 #f) (equal? expr2 #f)) #f]

[(or (equal? expr1 #t) (equal? expr2 #t)) #t]

[(equal? expr1 #f) expr2]

[(equal? expr2 #f) expr1]

[(list 'or expr1 expr2)]

))

;checks

(define-test-suite or-simp-suite

(check-equal? (or-simp #t #t) #t)

(check-equal? (or-simp #t #f) #t)

(check-equal? (or-simp #f #t) #t)

(check-equal? (or-simp #f #f) #f)

(check-equal? (or-simp #t '(and b g)) #t)

(check-equal? (or-simp '(not b) #t) #t)

(check-equal? (or-simp #f '(or a b)) '(or a b))

(check-equal? (or-simp '(and x y) #f) '(and x y))

(check-equal? (or-simp '(not a) '(not b)) '(or (not a) (not b)))

(check-equal? (or-simp 'a 'b) '(or a b))

)

(run-tests or-simp-suite 'verbose)

;bool-simp

;Simplfy a bolean expression.

;You should use the bool-eval given in Part 0 as a guide for the structure of this function.

;You are given an expression as input.

;Determine what kind of expression it is.

;The expressions can be constants, variables, NOT, AND, and OR.

;Use the bool-simp function to simplify the inputs to the expression recursively.

;AND and OR each have 2 operands, while NOT only has one.

;Once you have simplified the operands, pass them to the correct simplify helper function.

;

;A detailed example of how the function simplifies and expression is given below.

;(bool-simp '(or (not (and a #f)) (or #t (or c d))) )

;The top function is an or it has two operands

;op1 = (not (and a #f))

;op2 = (or #t (or c d)))

;We need to know what each of these operands simplifies two

;(bool-simp '(not (and a #f)))

;This is a not function, we need to simplify its one input.

;op1 = (and a #f)

;(bool-simp '(and a #f))

;This is an and function and has two operands

;op1 = a, this is a variable so (bool-simp 'a) returns a

;op2 = #f, this is a constant so (bool-simp #f) returns #f

;At this point we can use and-simp to see that

;(bool-simp '(and a #f)) = #f

;This allows us to move up the expression tree

;We go back to

;(bool-simp '(not (and a #f)))

;but we know know it is (not #f) because we have simplified (and a #f)

;This simplifies to (not #f)=#t

;We now know the first operands of the original input is #t.

;op1 = (not (and a #f)) = #t

;Now we simplify the second operand

;op2 = (or #t (or c d)))

;(bool-simp '(or #t (or c d))))

;This is an or expression

;op1 = #t

;op2 = (or c d)

;op1 is a constant so (bool-simp #t) returns #t

;op2 needs to be simplified

;(bool-simp '(or c d))

;The or has two operands

;op1 = c which is a variable and is returned by bool-simp

;op2 = d which is a variable and is returned by bool-simp

;or-simp has nothing to do

;(bool-simp '(or c d)) returns '(or c d)

;Now we know that both inputs are simplified and use or-simp in

;(bool-simp '(or #t (or c d))))

;If the first input is true, then the or is true

;(bool-simp '(or #t (or c d)))) = #t

;We now know the value of both operands.

;(bool-simp '(or (not (and a #f)) (or #t (or c d))) )

;op1 = (not (and a #f)) = #t

;op2 = (or #t (or c d))) = #t

;Calling (or-simp #t #t) tells us the simplification is #t

;(bool-simp '(or (not (and a #f)) (or #t (or c d))) ) returns #t

(define (bool-simp expr)

(cond

[(is-constant? expr) expr]

[(is-variable? expr) expr]

[(is-not? expr) (not-simp (bool-simp (op1 expr)))]

[(is-or? expr) (or-simp (bool-simp (op1 expr)) (bool-simp (op2 expr)))]

[(is-and? expr) (and-simp (bool-simp (op1 expr)) (bool-simp (op2 expr)))]

))

(define-test-suite bool-simp-suite

;Rules using not

(check-equal? (bool-simp '(not #t)) #f)

(check-equal? (bool-simp '(not #f)) #t)

(check-equal? (bool-simp '(not (and a b))) '(not (and a b)) )

;Rules using and

(check-equal? (bool-simp '(and #t #t)) #t)

(check-equal? (bool-simp '(and #t #f)) #f)

(check-equal? (bool-simp '(and #f #t)) #f)

(check-equal? (bool-simp '(and #f #f)) #f)

(check-equal? (bool-simp '(and (not a) #t)) '(not a))

(check-equal? (bool-simp '(and #t (or b c))) '(or b c))

(check-equal? (bool-simp '(and #f (not b))) #f)

(check-equal? (bool-simp '(and (or f g) #f)) #f)

(check-equal? (bool-simp '(and (or x y) (or a b))) '(and (or x y) (or a b)) )

;Rules using or

(check-equal? (bool-simp '(or #t #t)) #t)

(check-equal? (bool-simp '(or #t #f)) #t)

(check-equal? (bool-simp '(or #f #t)) #t)

(check-equal? (bool-simp '(or #f #f)) #f)

(check-equal? (bool-simp '(or #t (or x y))) #t)

(check-equal? (bool-simp '(or (or x y) #t)) #t)

(check-equal? (bool-simp '(or #f (not a))) '(not a))

(check-equal? (bool-simp '(or (and x y) #f)) '(and x y))

;Full Expressions

(check-equal? (bool-simp '(or (not (and a #f)) (or #t (or c d))) ) #t)

(check-equal? (bool-simp '(and (not #t) (or a b))) #f)

(check-equal? (bool-simp '(or (not (and #t b)) (and a (not c)))) '(or (not b) (and a (not c))))

(check-equal?

(bool-simp

'(or

(not

(or (and #t (or #t (and a b))) (not c))

);end of not 1

(not

(and (not #t) (and x y))

);end of not 2

);end of or

)

#t

)

(check-equal? (bool-simp '(not (not a))) 'a)

(check-equal? (bool-simp '(not (not (and a b)))) '(and a b))

(check-equal? (bool-simp '(or

(and (not (not #f)) (not (not (not #t))))

(not (not (and x y)))

);end of or

)

'(and x y)

)

)

(run-tests bool-simp-suite 'verbose)

#|

Part 2

This question is a proof by induction.

No code needs to be written.

The entire solution should be given in this comment area.

We want to prove that bool-simp produces an expression that is equivalent

to the original expression. I.E.

The evaluation of the original expression should provide the same answer

as evaluating the simplified answer.

For example,

We saw that

(bool-simp '(or (not (and a #f)) (or #t (or c d)))

returns

#t

(bool-eval '(or (not (and a #f)) (or #t (or c d))) env )

should be equal to

(bool-eval '#t env )

For any settings of the variables in environment provided

the variables occurring in the expressions are defined.

You will need to prove this is always true using induction.

The values of the variables in environment do not matter in this proof.

Your goal is to show that the induction is true for any environment.

What this means is you need to show that the two expressions being evaluated are the same.

You need to prove 4 theorems. Prove these in order, the solutions to

1-3 can be used in 4 and do not require induction.

Students should prove the following theorems:

1) (bool-eval '(not expr) env) = (bool-eval (not-simp expr) env)

To prove the above property, we need to prove '(not expr) = (not-simp expr). We give the proof by case analysis as below:

expr = constant:

expr = #t:

Leftside: '(not expr) = #f

Rightside: (not-simp expr)= #f

expr = #f:

Leftside: '(not expr)= #t

Rightside: (not-simp expr) = #t

expr is a variable:

Leftside: '(not expr)

Rightside: (not-simp expr) = '(not expr) by definition

expr is a AND expression:

Leftside: '(not expr)

Rightside: (not-simp expr) = '(not expr) by definition

expr is a OR expression:

Leftside: '(not expr)

Rightside: (not-simp expr) = '(not expr) by definition

expr is a NOT sub-expr:

Leftside: '(not expr) = '(not (not sub-expr)) = sub-expr

Rightside: (not-simp expr) = sub-expr by definition

2) (bool-eval '(and expr1 expr2) env) =

(bool-eval (and-simp expr1 expr2) env)

Similar to proof of (1), we need to prove '(and expr1 expr2) = (and-simp expr1 expr2)

in order to prove 2). We also give the proof by case analysis as below.

expr1 and expr2 both are constant:

expr1 = #t, expr2 = #t:

Leftside: '(and expr1 expr2) = #t

Rightside: (and-simp expr1 expr2) = #t

one of expr is constant:

expr1 = #t, expr2 = ALL(can be anything):

Leftside: '(and expr1 expr2) = '(and #t expr2) = expr2

Rightside: (and-simp expr1 expr2) = expr2

expr1 = #f, expr2 = ALL(can be anything):

Leftside: '(and expr1 expr2) = '(and #f expr2) = #f

Rightside: (and-simp expr1 expr2) = #f

expr1 = ALL(can be anything), expr2 = #t:

Leftside: '(and expr1 expr2) = '(and expr1 #t) = expr1

Rightside: (and-simp expr1 expr2) = expr1

expr1 = ALL(can be anything), expr2 = #f:

Leftside: '(and expr1 expr2) = '(and expr1 #f) = #f

Rightside: (and-simp expr1 expr2) = #f

expr1 is a varible, expr2 is a variable:

Leftside: '(and expr1 expr2)

Rightside: (and-simp expr1 expr2) = '(and expr1 expr2) by definition

expr1 is other cases (AND, OR, NOT sub-expr), expr2 is other cases (AND, OR, NOT sub-expr):

Leftside: '(and expr1 expr2)

Rightside: (and-simp expr1 expr2) = '(and expr1 expr2) by definition

3) (bool-eval '(or expr1 expr2) env) =

(bool-eval (or-simp expr1 expr2) env)

Similar to proof of (1) and (2), we need to prove '(or expr1 expr2) = (or-simp expr1 expr2)

in order to prove 3). We also give the proof by case analysis as below.

expr1 and expr2 both are constant:

expr1 = #f, expr2 = #f:

Leftside: '(or expr1 expr2) = #f

Rightside: (or-simp expr1 expr2) = #f

one of expr is constant:

expr1 = #t, expr2 = ALL(can be anything):

Leftside: '(or expr1 expr2) = '(or #t expr2) = #t

Rightside: (or-simp expr1 expr2) = #t

expr1 = #f, expr2 = ALL(can be anything):

Leftside: '(or expr1 expr2) = '(or #f expr2) = expr2

Rightside: (or-simp expr1 expr2) = expr2

expr1 = ALL(can be anything), expr2 = #t:

Leftside: '(or expr1 expr2) = '(or expr1 #t) = #t

Rightside: (or-simp expr1 expr2) = #t

expr1 = ALL(can be anything), expr2 = #f:

Leftside: '(or expr1 expr2) = '(or expr1 #f) = expr1

Rightside: (or-simp expr1 expr2) = expr1

expr1 is a varible, expr2 is a variable:

Leftside: '(or expr1 expr2)

Rightside: (or-simp expr1 expr2) = '(or expr1 expr2) by definition

expr1 is other cases (AND, OR, NOT sub-expr), expr2 is other cases (AND, OR, NOT sub-expr):

Leftside: '(or expr1 expr2)

Rightside: (or-simp expr1 expr2) = '(or expr1 expr2) by definition

4) (bool-eval expr env) = (bool-eval (bool-simp expr) env)

In order to prove (4), we need to prove expr = (bool-simp expr). We give the following case analysis and induction hyposis.

Assume that expr\_i = (bool-simp expr\_i) for expr\_i that has tree size less than expr in the representation of binary tree of boolean function.

Base cases:

expr = constant:

Leftside: expr

Rightside: (bool-simp expr) = expr

expr is a variable:

Leftside: expr

Rightside: (bool-simp expr) = expr

General cases:

expr is a NOT:

Leftside: expr is (not op1)

Rightside: (bool-simp expr) = (not-simp (bool-simp (op1 expr)))

= (not-simp op1) by IH

= (not op1) by above proved lemma (1)

expr is a AND:

Leftside: expr is (and op1 op2)

Rightside: (bool-simp expr) = (and-simp (bool-simp (op1 expr)) (bool-simp (op2 expr)))

= (and-simp op1 op2) by IH

= (and op1 op2) by above proved lemma (2)

expr is a OR:

Leftside: expr is (or op1 op2)

Rightside: (bool-simp expr) = (or-simp (bool-simp (op1 expr)) (bool-simp (op2 expr)))

= (or-simp op1 op2) by IH

= (or op1 op2) by above proved lemma (3)

|#

#|

Part 3

You have proved by induction that bool-simp produces an equivalent

expression. Next we need to prove that the expression is simplified.

First we write functions to test this.

If the simplifications worked correctly then one of the two cases should

be true about the simplified expression.

1.) There exist no constants nor double-negations in it

2.) The solution is a constant

Define the is-simplified function

If one of the above cases is true, then return true the expression is simplified.

If neither is true return false

|#

;no-constants

;Given an expression, return true if it contains no constants

;otherwise return false

;For example

;(no-constants '(and a (not c))) returns #t

;(no-constants '(or a (not #f)) returns #f

(define (no-constants? expr)

(cond

[(is-constant? expr) #f]

[(is-variable? expr) #t]

[(is-not? expr) (no-constants? (op1 expr))]

[else (and (no-constants? (op1 expr)) (no-constants? (op2 expr)))]

))

;Checks

(define-test-suite no-constants-suite

(check-equal? (no-constants? #f) #f)

(check-equal? (no-constants? #t) #f)

(check-equal? (no-constants? 'a) #t)

(check-equal? (no-constants? '(or a b)) #t)

(check-equal? (no-constants? '(and x y)) #t)

(check-equal? (no-constants? '(not g)) #t)

(check-equal? (no-constants? '(or (not (and #t a)) (or a b))) #f)

(check-equal? (no-constants? '(not (and a b) (or c d))) #t)

(check-equal? (no-constants? '(or (and (not a) (not b)) (or (and x y) #t)) ) #f)

)

(run-tests no-constants-suite 'verbose)

;no-double-negatives

;Returns #f if the expression contains a double negative

;(no-double-negatives '(not a)) returns t

;(no-double-negatives '(not (not b))) returns #f

(define (no-double-negatives? expr)

(cond

[(equal? null expr) #t]

[(is-constant? expr) #t]

[(is-variable? expr) #t]

[(is-constant? (op1 expr)) #t]

[(is-variable? (op1 expr)) #t]

[(is-and? expr) (and (no-double-negatives?(op1 expr)) (no-double-negatives?(op2 expr)))]

[(is-or? expr) (and (no-double-negatives?(op1 expr)) (no-double-negatives?(op2 expr)))]

[(and (is-not? expr) (is-not? (op1 expr))) #f]

[else #t]

))

;Checks

(define-test-suite no-double-negatives-suite

(check-equal? (no-double-negatives? #f) #t)

(check-equal? (no-double-negatives? #t) #t)

(check-equal? (no-double-negatives? 'a) #t)

(check-equal? (no-double-negatives? '(not (not a))) #f)

(check-equal? (no-double-negatives? '(and (or (not a) (not b)) (not (not c)))) #f)

(check-equal? (no-double-negatives? '(or (and (not (not a)) b) c)) #f)

(check-equal? (no-double-negatives? '(or (and (not a) (not b)) (or (not x) (not y)))) #t)

)

(run-tests no-double-negatives-suite 'verbose)

;is-simplified?

;Given an expression determine if it is simplified

;A simplified expression is a constant or contains no constants

(define (is-simplified? expr)

(cond

[(and (or (is-constant? expr) (no-constants? expr)) (no-double-negatives? expr)) #t]

[else #f]

))

;Checks

(define-test-suite is-simplified-suite

(check-equal? (is-simplified? #f) #t)

(check-equal? (is-simplified? #t) #t)

(check-equal? (is-simplified? 'a) #t)

(check-equal? (is-simplified? '(or a b)) #t)

(check-equal? (is-simplified? '(and x y)) #t)

(check-equal? (is-simplified? '(not g)) #t)

(check-equal? (is-simplified? '(or (not (and #t a)) (or a b))) #f)

(check-equal? (is-simplified? '(not (and a b) (or c d))) #t)

(check-equal? (is-simplified? '(or (and (not a) (not b)) (or (and x y) #t))) #f)

(check-equal? (is-simplified? #f) #t)

(check-equal? (is-simplified? #t) #t)

(check-equal? (is-simplified? 'a) #t)

(check-equal? (is-simplified? '(not (not a))) #f)

(check-equal? (is-simplified? '(and (or (not a) (not b)) (not (not c)))) #f)

(check-equal? (is-simplified? '(or (and (not (not a)) b) c)) #f)

(check-equal? (is-simplified? '(or (and (not a) (not b)) (or (not x) (not y)))) #t)

(check-equal? (is-simplified? '(or (and (not a) (not b)) (or (not #f) (not y)))) #f)

(check-equal? (is-simplified? '(or (and (not a) (not (not b))) (or (not #f) (not y)))) #f)

)

(run-tests is-simplified-suite 'verbose)

#|

Part 4

Prove by induction that (is-simplified? (bool-simp expr))

Assume (is-simplified? (bool-simp expr\_i)) = #t for all expr\_i that size less than expr by induction hypnosis.

(is-simplified? (bool-simp expr))

Base cases:

expr = constant:

(is-simplified? (bool-simp expr))

=(is-simplified? constant) by def of bool-simp and is-constant

= #t by def of is-simplified

expr is a variable:

(is-simplified? (bool-simp expr))

=(is-simplified? variable) by def of bool-simp and is-variable

= #t by def of is-simplified

General cases:

expr is a NOT:

Assume that (is-simplified? (bool-simp expr)) = #t by IH

=(is-simplified? (not-simp (bool-simp (op1 expr)))) by def of bool-simp

Is-simplified? Can branch off into 3 possibilities:

1. is-constant?

(is-constant?(bool-simp expr)) = #t

Then (is-constant? (not-simp (bool-simp expr))) = #t by def of not-simp

Then (is-simplified? (not-simp (bool-simp expr))) = #t by def of is-simplified?

1. no-constants?

assuming that is-constant? Is #f, then

(no-constants?(bool-simp expr)) = #t by previous assumption

Then (no-constants? (not-simp (bool-simp expr))) = #t by def of not-simp

Then (is-simplified? (not-simp (bool-simp expr))) = #t by def of is-simplified? And IH

1. no-double-negatives?

Given that expression is either a constant or has no constants:

(no-double-negatives?(bool-simp expr)) = #t

Then (no-double-negatives?((not-simp(bool-simp expr))) = #t by def of not-simp

Then (is-simplified?((not-simp(bool-simp expr))) = #t by def of is-simplified and IH

expr is AND:

Assume that (is-simplified? (bool-simp expr)) = #t by IH

Then (is-simplified?(bool-simp (op1 expr))) ^ (is-simplified?(bool-simp(op2 expr))) = #t

=(is-simplified? (and-simp (bool-simp (op1 expr)) (bool-simp(op2 expr)))) by def bool-simp

=(is-simplified? (bool-simp(op1 expr))) ^ (is-simplified? (bool-simp(op2 expr))) by def of bool-simp

3 possible cases arise from is-simplified?:

1. Is-constant?

(is-constant? (bool-simp expr) = #t

= (is-constant? (and-simp (bool-simp (op1 expr)) (bool-simp (op2 expr))) by def of bool-simp

Then (is-simplified? (and-simp(bool-simp (op1 expr)) (bool-simp(op2 expr)) = #t by def of is-simplified and IH

1. No-constants?

(no-constants?(bool-simp expr))

=(no-constants? (bool-simp (op1 expr))) ^ (no-constants?(bool-simp (op2 expr)))

=(and (no-constants(bool-simp(op1 expr)) (no-constants?(bool-simp(op2 expr))) by def of and

=(and #t #t) = #t by IH

Then (is-simplified?(and-simp (bool-simp (op1 expr)) (bool-simp (op2 expr))) = #t

1. No-double-negatives?

Given that expression is either a constant or has no constants:

(no-double-negatives?(bool-simp expr)) = #t

Then (no-double-negatives?((and-simp (bool-simp (op1 expr)) (bool-simp (op2 expr))) = #t by def of and-simp

Then (is-simplified?((and-simp(bool-simp (op1 expr)) (bool-simp (op2 expr))) = #t by def of is-simplified and IH

expr is OR:

Assume that (is-simplified? (bool-simp expr)) = #t by IH

Then (is-simplified?(bool-simp (op1 expr))) or (is-simplified?(bool-simp(op2 expr))) = #t

=(is-simplified? (or-simp (bool-simp (op1 expr)) (bool-simp(op2 expr)))) by def bool-simp

=(is-simplified? (bool-simp(op1 expr))) or (is-simplified? (bool-simp(op2 expr))) by def of bool-simp

3 possible cases arise from is-simplified?:

1. Is-constant?

(is-constant? (bool-simp expr) = #t

= (is-constant? (or-simp (bool-simp (op1 expr)) (bool-simp (op2 expr))) by def of bool-simp

Then (is-simplified? (or-simp(bool-simp (op1 expr)) (bool-simp(op2 expr)) = #t by def of is-simplified and IH

1. No-constants?

(no-constants?(bool-simp expr))

=(no-constants? (bool-simp (op1 expr))) or (no-constants?(bool-simp (op2 expr)))

=(or (no-constants?(bool-simp(op1 expr)) (no-constants?(bool-simp(op2 expr))) by def of and

=(or #t #t) = #t by IH

Then (is-simplified?(or-simp (bool-simp (op1 expr)) (bool-simp (op2 expr))) = #t

1. No-double-negatives?

Given that expression is either a constant or has no constants:

(no-double-negatives?(bool-simp expr)) = #t

Then (no-double-negatives?((or-simp (bool-simp (op1 expr)) (bool-simp (op2 expr))) = #t by def of and-simp

Then (is-simplified?((or-simp(bool-simp (op1 expr)) (bool-simp (op2 expr))) = #t by def of is-simplified and IH

|#