CISP 440

Austin Smothers

Homework 11.42

/\*

Argument Validator

Austin Smothers April 2017

This program builds on code provided by Professor Ross

Generates an argument in table format and checks the argument for validity.

Given the propositions:

p = "The moon is made of cheese."

q = "Winter is coming."

The following argument is VALID:

"The moon is made of cheese or winter is coming. The moon is not made of cheese. Therefore, winter is coming."

Symbolically:

P1: pVq

P2: !p

Q: q

As stored in Argument table for validation:

P1 P2 Q

p q pVq !p q

0 0 0 1 0 //row0

0 1 1 1 1 //row1

1 0 1 0 0 //row2

1 1 1 0 1 //row3

//0 1 2 3 4

HOWEVER, The following argument is INVALID:

"The moon is made of cheese or winter is coming. The moon is made of cheese. Therefor, winter is coming."

Symbolically:

P1: pVq

P2: p

Q: q

As stored in Argument table for validation:

P1 P2 Q

p q pVq p q

0 0 0 0 0 //row0

0 1 1 0 1 //row1

1 0 1 1 0 //row2 INVALID HERE

1 1 1 1 1 //row3

\*/

#include <stdio.h>

#include <iostream>

using namespace std;

#pragma warning( disable : 4805)

#pragma warning( disable : 4800)

// Required logic function prototypes, in operation precedence order.

// See previous homework for truth table definitions.

bool NOT(bool p); //not

bool AND(bool p, bool q); //and

bool OR(bool p, bool q); //inclusive or

bool XOR(bool p, bool q); //exclusive or

bool IMP(bool p, bool q); //implication

bool BCN(bool p, bool q); //bicondition

bool MAY(bool p, bool q); //maybe

bool BEC(bool p, bool q); //because

/\*

The Argument table

Column format is lil propositions, big (compound) propositions, conclusion:

p, q, ... z, P1, P2, ... PN, Q

Row format is all pqr value permutations in binary count order:

00

01

10

11

etc...

\*/

#define LOTS 42

bool Argument[LOTS][LOTS];

int NLilProps; // how many lil propositions, p thru z

int NBigProps; // how many big propositions, P1 thru PN

/\*

Calculates base raised to the power exp

\*/

int my\_pow(int base, int exp)

{

int x = 1;

for (int i = 0; i < exp; i++)

x \*= base;

return x;

}

// Does what its name suggests

void PrintArgument()

{

for (int row = 0, rows = my\_pow(2, NLilProps); row < rows; row++)

{

for (int col = 0, cols = NLilProps + NBigProps; col < cols; col++)

printf("%d ", Argument[row][col]);

printf("\n");

}

}

/\*

Implements OR logical operation.

Why? To show another a way to do this kinda thing.

Has a nice function syntax, makes writing long logic expressions syntactically consistent.

Makes other logic operations easy to implement.

Contains efficient code tricks.

\*/

bool OR(bool p, bool q)

{

// What OR means

bool T[4] = { 0, 1, 1, 1 };

// calculate index with bit operations for more fasterness

// then do a lookup, also fast

return T[(p << 1) | q];

}

/\*

Implements NOT logical operation

\*/

bool NOT(bool p)

{

// What NOT means

bool T[2] = { 1, 0 };

// lookup

return T[p];

}

/\*

Implements AND logical operation

\*/

bool AND(bool p, bool q)

{

/\*despite sample operations being "more efficient"

I still prefer to stick with classic methods\*/

return (p && q);

}

/\*

Implements XOR logical operation

\*/

bool XOR(bool p, bool q)

{

return (!(p == q));

}

/\*

Implements IMPLICATION logical operation

\*/

bool IMP(bool p, bool q)

{

return ((!p) || (q));

}

/\*

Implements BICONDITIONAL logical operation

\*/

bool BCN(bool p, bool q)

{

return (p == q);

}

/\*

Implements MAYBE logical operation

\*/

bool MAY(bool p, bool q)

{

return (q && (!p));

}

/\*

Implements BECAUSE logical operation

\*/

bool BEC(bool p, bool q)

{

return p;

}

/\*

The premise of this function is to try to prove that the argument is not invalid.

If the argument is not invalid, then it must be valid.

\*/

bool IsValid()

{

/\*

Using the print function's loop to systematically check all rows & cols of the

argument.

\*/

for (int row = 0, rows = my\_pow(2, NLilProps); row < rows; row++)

{

for (int col = NLilProps, cols = NLilProps + NBigProps; col < cols; col++)

{

/\*

First If/Else hurtle

The first condition evaluates to true only if two consecutive premises do not both evaluate to true (if all premises aren't true, the conclusion can't be invalid).

The second condition evaluates to true only if any two premises are being compared; it is false if the last premise is being compared to the conclusion

Combined: If, while checking the premises, any two consecutive premises are not both true, then move on to the next row (via break)

\*/

if ((NOT(AND(Argument[row][col], Argument[row][col + 1]))

&& (!(cols == col + 2))))

break;

/\*

Second If/Else hurtle

The first condition evaluates to true only if the last premise is being compared to the conclusion (you can't determine validity if you haven't found all premises to be true, which the first if/else ensures)

The second condition evaluates to true only if the last premise and the conclusion are not both true (given that, if any premise weren't true, the first if/else would have break'd to the next row, this condition can only be met if the last premise is true, but the conclusion is false)

The third condition makes sure that the final premise is, in fact, true. If the final premise is false, then the row in question cannot be invalid.

Combined: Given that all premises are true, if the conclusion is false, return false, as the argument must be invalid

\*/

else if ((cols == col + 2) &&

NOT(AND(Argument[row][col], Argument[row][col + 1]))

&& Argument[row][col])

{

cout << "Error! Row " << row << " is not valid!\n";

return false;

}

/\*

If both of the above statements evaluate to false, then both previously compared elements must have been true. In such an event, iterate the columns/rows as necessary and keep comparing

\*/

}

/\*

If the entire Argument array is found to be valid, return true, as the argument must be valid, as the above if/else checks have proven it to be not invalid.

\*/

}

return true;

}

/\*

"I'd like to have an argument, please."

"I'm not allowed to argue unless you've paid."

"...No it isn't."

\*/

void main(void)

{

/\*

ARGUMENT 1

\*/

cout << "Argument 1:" << endl;

// A fancy interactive parser would determine these values from user console input

NLilProps = 2;

NBigProps = 3; // includes Q

int rows = my\_pow(2, NLilProps); // how many rows

int row = 0; // current row

while (row < rows)

{

bool p = row & 1; bool q = row & 2;

// NOTE: An INTERACTIVE version could send string expressions to an evaluator here

bool P1 = OR(p, q); // something like P1 = eval(strP1, p, q); where str1 is "pVq" entered by the user.

bool P2 = NOT(p);

bool Q = q;

// Fill the argument table

Argument[row][0] = p;

Argument[row][1] = q;

Argument[row][2] = P1;

Argument[row][3] = P2;

Argument[row][4] = Q;

row++;

}

// print out each argument

printf(" P1 P2 Q\n");

printf("p q pVq !p q\n\n");

PrintArgument();

// "Open the pod bay doors please HAL"

// References are from the classic movie "2001: A Space Odyssey" REQUIRED viewing for all CS majors.

if (IsValid())

printf("Your argument is valid, Dave. I enjoy working with humans.\n");

else

printf("I'm sorry Dave, but I'm afraid your argument is invalid.\n");

/\*

ARGUMENT 2

\*/

cout << endl << "Argument 2:" << endl;

// A fancy interactive parser would determine these values from user console input

NLilProps = 3;

NBigProps = 3; // includes Q

rows = my\_pow(2, NLilProps); // how many rows

row = 0; // current row

while (row < rows)

{

bool p = row & 1; bool q = row & 2; bool r = row & 4;

bool P1 = IMP(p, r);

bool P2 = IMP(p, q);

bool Q = IMP(p, AND(r, q));

// Fill the argument table

Argument[row][0] = p;

Argument[row][1] = q;

Argument[row][2] = r;

Argument[row][3] = P1;

Argument[row][4] = P2;

Argument[row][5] = Q;

row++;

}

// print out each argument

printf(" P1 P2 Q\n");

printf("p q r p->r p->q p->(rAq)\n\n");

PrintArgument();

// "Open the pod bay doors please HAL"

// References are from the classic movie "2001: A Space Odyssey" REQUIRED viewing for all CS majors.

if (IsValid())

printf("Your argument is valid, Dave. I enjoy working with humans.\n");

else

printf("I'm sorry Dave, but I'm afraid your argument is invalid.\n");

/\*

ARGUMENT 3

\*/

cout << endl << "Argument 3:" << endl;

// A fancy interactive parser would determine these values from user console input

NLilProps = 2;

NBigProps = 2; // includes Q

rows = my\_pow(2, NLilProps); // how many rows

row = 0; // current row

while (row < rows)

{

bool p = row & 1; bool q = row & 2; bool r = row & 4;

bool P1 = AND(p, NOT(p));

bool Q = q;

// Fill the argument table

Argument[row][0] = p;

Argument[row][1] = q;

Argument[row][2] = P1;

Argument[row][3] = Q;

row++;

}

// print out each argument

printf(" P1 Q\n");

printf("p q pA!p q\n\n");

PrintArgument();

// "Open the pod bay doors please HAL"

// References are from the classic movie "2001: A Space Odyssey" REQUIRED viewing for all CS majors.

if (IsValid())

printf("Your argument is valid, Dave. I enjoy working with humans.\n");

else

printf("I'm sorry Dave, but I'm afraid your argument is invalid.\n");

/\*

ARGUMENT 4

\*/

cout << endl << "Argument 4:" << endl;

// A fancy interactive parser would determine these values from user console input

NLilProps = 4;

NBigProps = 3; // includes Q

rows = my\_pow(2, NLilProps); // how many rows

row = 0; // current row

while (row < rows)

{

bool p = row & 1; bool q = row & 2; bool r = row & 4; bool s = row & 8;

bool P1 = AND(IMP(p, q), IMP(r, s));

bool P2 = OR(p, r);

bool Q = OR(q, s);

// Fill the argument table

Argument[row][0] = p;

Argument[row][1] = q;

Argument[row][2] = r;

Argument[row][3] = s;

Argument[row][4] = P1;

Argument[row][5] = P2;

Argument[row][6] = Q;

row++;

}

// print out each argument

printf(" P1 P2 Q\n");

printf("p q r s p->q pvr qvs\n"

" Ar->s\n\n");

PrintArgument();

// "Open the pod bay doors please HAL"

// References are from the classic movie "2001: A Space Odyssey" REQUIRED viewing for all CS majors.

if (IsValid())

printf("Your argument is valid, Dave. I enjoy working with humans.\n");

else

printf("I'm sorry Dave, but I'm afraid your argument is invalid.\n");

/\*

ARGUMENT 5

\*/

cout << endl << "Argument 5:" << endl;

// A fancy interactive parser would determine these values from user console input

NLilProps = 3;

NBigProps = 3; // includes Q

rows = my\_pow(2, NLilProps); // how many rows

row = 0; // current row

while (row < rows)

{

bool p = row & 1; bool q = row & 2; bool r = row & 4;

bool P1 = XOR(p, q);

bool P2 = BCN(q, r);

bool Q = r;

// Fill the argument table

Argument[row][0] = p;

Argument[row][1] = q;

Argument[row][2] = r;

Argument[row][3] = P1;

Argument[row][4] = P2;

Argument[row][5] = Q;

row++;

}

// print out each argument

printf(" P1 P2 Q\n");

printf("p q r p+q q<->r r\n\n");

PrintArgument();

// "Open the pod bay doors please HAL"

// References are from the classic movie "2001: A Space Odyssey" REQUIRED viewing for all CS majors.

if (IsValid())

printf("Your argument is valid, Dave. I enjoy working with humans.\n");

else

printf("I'm sorry Dave, but I'm afraid your argument is invalid.\n");

/\*

ARGUMENT 6

\*/

cout << endl << "Argument 6:" << endl;

// A fancy interactive parser would determine these values from user console input

NLilProps = 3;

NBigProps = 3; // includes Q

rows = my\_pow(2, NLilProps); // how many rows

row = 0; // current row

while (row < rows)

{

bool p = row & 1; bool q = row & 2; bool r = row & 4;

bool P1 = MAY(p, r);

bool P2 = BEC(p, q);

bool Q = BCN(q, XOR(r, p));

// Fill the argument table

Argument[row][0] = p;

Argument[row][1] = q;

Argument[row][2] = r;

Argument[row][3] = P1;

Argument[row][4] = P2;

Argument[row][5] = Q;

row++;

}

// print out each argument

printf(" P1 P2 Q\n");

printf("p q r p?r p@q q<->r+p\n\n");

PrintArgument();

// "Open the pod bay doors please HAL"

// References are from the classic movie "2001: A Space Odyssey" REQUIRED viewing for all CS majors.

if (IsValid())

printf("Your argument is valid, Dave. I enjoy working with humans.\n");

else

printf("I'm sorry Dave, but I'm afraid your argument is invalid.\n");

}



