

Gregory Maldonado
Project 2: Clojure Symbolic Simplification and Evaluation

Pre-Processing:

The simplification of a symbolic expression can be done with four Clojure functions:

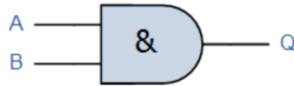
- **defn** and-simplify [exp]
- **defn** or-simplify [exp]
- **defn** not-simplify [exp]
- **defn** demorgans [n new-coll coll] : (for special cases which will be discussed later)

All the methods defined above are recursively defined with the exception of 'not-simplify.' A function: **defn** simplify-exp [exp] was created as a utility function to recursively simplify expressions that have nested expressions, i.e '(and x (or true false)). The 'simplify-exp' function will simplify '(or true false) first and then use the return expression to simplify the outer expression.

- ⇒ '(and x (or true false))
- ⇒ '(and x true)
- ⇒ x

AND Boolean Operator:

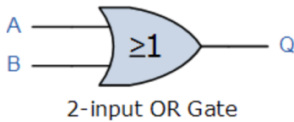
The function **defn** and-simplify [exp] allows the program to simplify expressions that contains the AND Boolean operator. In an AND expression, both Booleans have to be true in order for the final evaluation to be true. If there is a variable within the expression then the returned evaluation will include the variable because the final evaluation depends on the value of that variable.

Symbol	Truth Table		
 2-input AND Gate	A	B	Q
	0	0	0
	0	1	0
	1	0	0
	1	1	1
Boolean Expression $Q = A.B$		Read as A AND B gives Q	

The algorithm for the 'and-simplify' function first checks the type of each of the arguments in the expression. If both arguments are Booleans, then the function can just evaluate the expression and return another Boolean. If any of the arguments are type symbol, then the function will have to return a final evaluation containing the variable(s).

OR Boolean Operator:

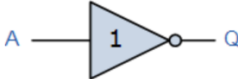
The function `defn or-simplify [exp]` allows the program to simplify expressions that contains the OR Boolean operator. In an OR expression, one of the Booleans has to be true in order for the final evaluation to be true. If there are two variable within the expression then the returned evaluation will be '(or x y). If there is only one variable and the other Boolean is True, then the returned evaluation will be True but if the Boolean is false then the final evaluation depends on the variable.

Symbol	Truth Table		
 2-input OR Gate	A	B	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	1
Boolean Expression $Q = A + B$		Read as A OR B gives Q	

The algorithm for the 'or-simplify' function first checks the type of each of the arguments in the expression. If both arguments are Booleans, then the function can just evaluate the expression and return another Boolean. If one of the types are type symbol and the other type is a Boolean, then the algorithm will check the value of the Boolean and can determine whether to return True or the variable. If both of the arguments are type symbol, then the function will have to return a final evaluation containing the variables.

NOT Boolean Operator:

The function `defn or-simplify [exp]` allows the program to simplify expressions that contains the NOT Boolean operator. In a NOT expression, the Boolean value is negated. If the expression contains a Boolean, then the negated Boolean will be returned: '(not True) returns False. If the expression contains a variable, then the function will return '(not x) because it depends on the Boolean of the variable.

Symbol	Truth Table	
 Inverter or NOT Gate	A	Q
	0	1
	1	0
Boolean Expression $Q = \text{NOT } A \text{ or } \bar{A}$		Read as inversion of A gives Q

The algorithm for the 'not-simplify' function first checks the type of the argument in the expression. If the argument is a Boolean, then the function can just evaluate the expression and

return the negated Boolean. If the argument is type symbol then the function will return a list '(not x).

Putting The Functions Together:

A. De-Morgan's Law

De-Morgan's Law are special evaluations for the expressions:

- '(not (and A B))
- '(not (or A B))

These expressions evaluate to:

- '(not (and A B)) => '(or (not A) (not B))
- '(not (or A B)) => '(and (not A) (not B))

The function **defn** demorgans [n new-coll coll] is recursive where the parameter n is the number of lists within the expression. Each recursive call, the function negates each Boolean and for the next recursive call, n is decreases by one. When n is zero, that means each Boolean has been negated and the function can return the final evaluated expression.

B. Further Expression Evaluation

The function **defn** evalexp [l m] is recursive and takes an expression and a map of variables to Booleans. The function does a deep substitute on the expression with the map of variables and returns the new substituted expression.

```
(evalexp '(and x (or x (and y (not z)))) '{x false, z true})
```

Returns -> '(and false (or false (and y (not true))))

-> False