

Function 1, findVarIds:

This function returns all the distinct variables in a propositional formula. It is useful for being passed to genVarAsgns to calculate all variable possibilities

It works by recurring on any of the more complex statements. Such as and, or, iff, etc. The base case is either a Const or Var at which point it will add them to the list. An essential part of the function is the nub[] placed on all combination cases. This is to insure there are no duplicates among the output.

Function 2, genVarAsgns:

This one was among the most complex to implement as it was the hardest to think of in abstract. The concept is simple enough, take a list of variables and generate all 2^N possibilities of True False values.

In execution however, I had never done anything like this in Haskell. The way it ultimately ended up working was Tree-like in its creation. I went with this because a tree is the best way I could picture a 2^N recursion. The base case is simply an empty map when there are no further variables in the List given. On execution however we start with 2 branches of the first variable lets say for this example list [x,y,z]. So it makes a True False branch for x before recurring and making a True false branch for y for both of those x branches. Then finally a True false branch for z on every one of those y branches before terminating on the base case. This will result in 2^N total permutations. We then concatenate these permutations with the ++ and end with a full map of all 2^N possibilities. I encountered quite a challenge with this function and the syntax took me multiple hours to get just right.

I even had to create a Notepad file to understand it. Pictured below.

genVarAsgns:

Given n variable names 2^n variable assignments possible

example: x, y, z with T-F

Only x: T(x), F(x)

x and y: T(x) T(y), T(x) F(y), F(x) T(y), F(x) F(y)

x y and z:

T(x) T(y) T(z), T(x) F(y) F(z), T(x) F(y) T(z),

T(x) T(y) F(z), F(x) T(y) T(z), F(x) T(y) F(z),

F(x) F(y) T(z), F(x) F(y) F(z)

Function 3, eval:

This one is rather complex, though not quite as complex as genVarAsgns from a design perspective.

Rather than a tree like approach, this function is conceptualized more like a flowchart that ultimately works its way backwards to yield a result. Like findVarIds this one has both Const and Var as a base case.

Const is simply evaluated as whatever Bool it is attached to, while Var is more complex. Var is evaluated using the lookup function of Map. Given a var's Id it will find its associated bool value in the Map. If for some reason there is not one, it will yield False otherwise whatever the given bool is will be retrieved.

From there it is simply a logical extension of the same process used in findVarIds with higher-order

operations such as and, or, iff, etc. Resulting in recursion before being evaluated logically. For example, if variable x is True and variable y is False after lookup, then the recursion from And is finished and it evaluates True && False which is False. I chose this approach as I found the concept of isolating a variable similar to findVarIds so I approached it the same way. The only difference being a more complex var base case due to being a key, value pair rather than a single variable.

Function 4, sat:

From here we start getting into functions that utilize the previous ones. Whereas all 3 previous functions were independent. This one uses a chain of all 3 with some prelude and map functions to give a resulting Bool.

Firstly it feeds a Prop formula to findVarIds, this then generates a list for genVarAsgns to use which will yield a list of Mapped vars and bools. We then map the function eval loaded with the given propositional formula over all these possibilities. This produces a list of either True or False's generated by the possibilities. Elem True will see if any of these are True then sat will produce True, otherwise False.

Function 5, readFormula:

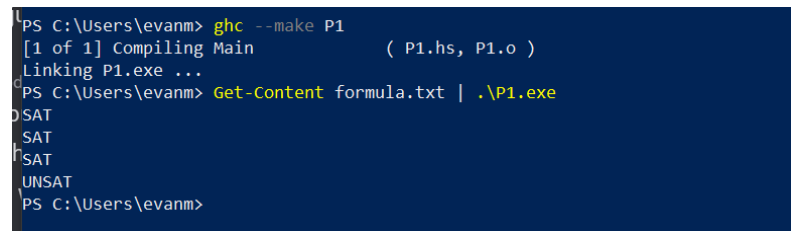
A very simple function. The hint in the P1.pdf was very helpful and is the reason I chose this implementation. It simply takes a string as input and uses the Read function/module from deriving to assign it the Prop type. Very little going on with this one. Though it is quite important as without it we would have no way of converting input text to the Prop data type to be used in the other functions.

Function 6, checkFormula:

Finally for functions, we have checkFormula. Which utilizes readFormula in conjunction with sat to produce either SAT or UNSAT depending on whether sat() yields True or False respectively. Most of the work is done by sat() calling the 3 other functions while readFormula just converts the given formula from String to Prop. checkFormula takes the Bool yielded by sat() and turns it into a String ("SAT" or "UNSAT" as previously stated). Originally it was slightly more complex with a case statement, but Visual Studio informed me I could simplify it, resulting in what is present now.

Main:

Last but not least, we have the Main. This is again quite simple, we merely load in the given file with getContents, break up the Strings by newline with lines. Finally, we map checkFormula over all the list of separated strings and output the result given by checkFormula to console. As pictured below.



```
PS C:\Users\evanm> ghc --make P1
[1 of 1] Compiling Main             ( P1.hs, P1.o )
Linking P1.exe ...
PS C:\Users\evanm> Get-Content formula.txt | .\P1.exe
SAT
SAT
SAT
UNSAT
PS C:\Users\evanm>
```

Formula.txt contains the sample given in the P1.pdf handout. I managed to put this together with help from the lecture slides (Particularly Slide 12 of 8-IO2). There are other ways to do this, but this is the one I found most simple, so I went with it.