Logic, Languages and Programming

Jan van Eijck
CWI & ILLC, Amsterdam

Guest Lecture Minor Programming
January 20, 2014

Abstract

This lecture will combine the topics of the title in various ways. First I will show that logic is part of every programming language, in the form of boolean expressions. Next, we will analyze the language of boolean expressions a bit, looking both at syntax and semantics. If the language of boolean expressions is enriched with quantifiers, we move from propositional logic to predicate logic. I will discuss how the expressions of that language can describe the ways things are. Next, I will say something about model checking, and about the reverse side of the expressive power of predicate logic. I end with the use of logic to describe invariants of algorithms. In the course of the lecture I will connect everything with (functional) programming, and you will be able to pick up some Haskell as we go along.

Every Programming Language Uses Logic

From a Java Exercise: suppose the value of b is false and the value of x is 0. Determine the value of each of the following expressions:

```
b && x == 0
b || x == 0
!b && x == 0
!b || x == 0
b && x != 0
b || x != 0
!b && x != 0
!b || x != 0
```

Question 1 *What are the answers to the Java exercise?*

The Four Main Ingredients of Imperative Programming

Assignment Put number 123 in location x

Concatenation First do this, next do that

Choice If this condition is true then do this, else do that.

Loop As long as this condition is true, do this.

The conditions link to a language of logical expressions that has negation, conjunction and disjunction.

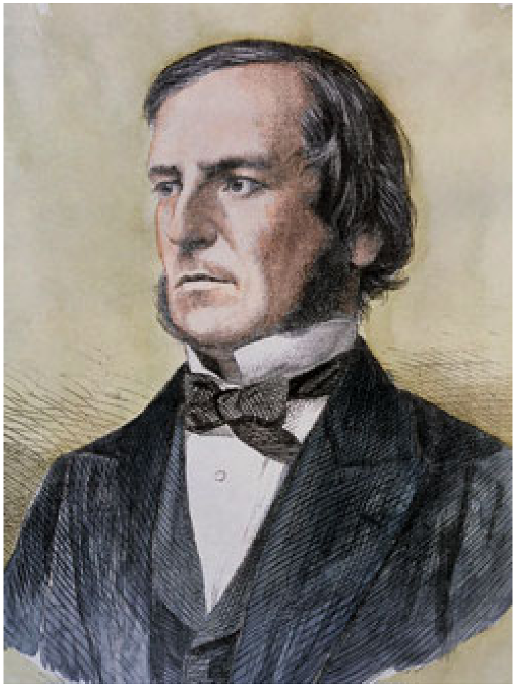
Let’s Talk About Logic

Talking about logic means: talking about a logical language.

The language of expressions in programming is called Boolean logic or propositional logic.

Context free grammar for propositional logic:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi.$$


George Boole (1815 – 1864)

Literate Programming, in Haskell

See http://www.haskell.org

```
module Logic

where

import Data.List
import Data.Char
```

A Datatype for Formulas

```
type Name = Int

data Form = Prop Name
          | Neg  Form
          | Cnj [Form]
          | Dsj [Form]
          | Impl Form Form
          | Equiv Form Form
          deriving Eq
```

This looks almost the same as the grammar for propositional logic.

Example Formulas

```
p = Prop 1
q = Prop 2
r = Prop 3

form1 = Equiv (Impl p q) (Impl (Neg q) (Neg p))
form2 = Equiv (Impl p q) (Impl (Neg p) (Neg q))
form3 = Impl (Cnj [Impl p q, Impl q r]) (Impl p r)
```

Validity

```
*Logic> form1
((1==>2)<=>(-2==>-1))
*Logic> form2
((1==>2)<=>(-1==>-2))
*Logic> form3
(*((1==>2),(2==>3))==>(1==>3))
```

Question 2 *A formula that is always true, no matter whether its proposition letters are true, is called valid. Which of these three formulas are valid?*

Validity

```
*Logic> form1
((1==>2)<=>(-2==>-1))
*Logic> form2
((1==>2)<=>(-1==>-2))
*Logic> form3
(*((1==>2),(2==>3))==>(1==>3))
```

Question 2 *A formula that is always true, no matter whether its proposition letters are true, is called valid. Which of these three formulas are valid?*

Answer: `form1` and `form3`.

Proposition Letters (Indices) Occurring in a Formula

```
propNames :: Form -> [Name]
propNames = sort.nub.pnames where
  pnames (Prop name) = [name]
  pnames (Neg f)   = pnames f
  pnames (Cnj fs) = concat (map pnames fs)
  pnames (Dsj fs) = concat (map pnames fs)
  pnames (Impl f1 f2) = concat (map pnames [f1,f2])
  pnames (Equiv f1 f2) =  concat (map pnames [f1,f2])
```

To understand what happens here, we need to learn a bit more about functional programming.

Question 3 *Why is it important to know which proposition letters occur in a formula?*

Proposition Letters (Indices) Occurring in a Formula

```
propNames :: Form -> [Name]
propNames = sort.nub.pnames where
  pnames (Prop name) = [name]
  pnames (Neg f)   = pnames f
  pnames (Cnj fs) = concat (map pnames fs)
  pnames (Dsj fs) = concat (map pnames fs)
  pnames (Impl f1 f2) = concat (map pnames [f1,f2])
  pnames (Equiv f1 f2) =  concat (map pnames [f1,f2])
```

To understand what happens here, we need to learn a bit more about functional programming.

Question 3 *Why is it important to know which proposition letters occur in a formula?*

Answer: Because the truth of the formula depends on the truth/falsity of these.

Type Declarations

```
propNames :: Form -> [Name]
```

This is a type declaration or type specification. It says that `propNames` is a function that takes a `Form` as an argument, and yields a list of `Names` as a value. `[Name]` is the type of a list of `Names`.

This function is going to give us the names (indices) of all proposition letters that occur in a formula.

map, concat, sort, nub

If you use the command `:t` to find the types of the predefined function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

This tells you that map is a higher order function: a function that takes other functions as arguments. map takes a function of type `a -> b` as its first argument, and yields a function of type `[a] -> [b]` (from lists of `a`s to lists of `b`s).

In fact, the function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

Thus `map pnames fs` is the command to apply the `pnames` function to all members for `fs` (a list of formulas) and collect the results in a new list.

Question 4 *What is the type of this new list?*

map, concat, sort, nub

If you use the command `:t` to find the types of the predefined function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

This tells you that map is a higher order function: a function that takes other functions as arguments. map takes a function of type `a -> b` as its first argument, and yields a function of type `[a] -> [b]` (from lists of `a`s to lists of `b`s).

In fact, the function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

Thus `map pnames fs` is the command to apply the `pnames` function to all members for `fs` (a list of formulas) and collect the results in a new list.

Question 4 *What is the type of this new list?*

Answer: a list of lists of names: `[[Name]]`.

Mapping

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here `(^2)` is a short way to refer to the squaring function.

Mapping

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here `(^2)` is a short way to refer to the squaring function.

Here is a definition of `map`, including a type declaration.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : map f xs
```

In Haskell, the colon `:` is used for putting an element in front of a list to form a new list.

Question 5 *What does* `(x:xs)` *mean? Why does* `map` *occur on the righthand-side of the second equation?*

Mapping

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here `(^2)` is a short way to refer to the squaring function.

Here is a definition of `map`, including a type declaration.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : map f xs
```

In Haskell, the colon `:` is used for putting an element in front of a list to form a new list.

Question 5 *What does* `(x:xs)` *mean? Why does* `map` *occur on the righthand-side of the second equation?*

Answer: `(x:xs)` is the pattern of a non-empty list. The call on the righthand side of the second equation is an example of recursion.

List Concatenation: ++

++ is the operator for concatenating two lists. Look at the type:

```
*Logic> :t (++)
(++) :: forall a. [a] -> [a] -> [a]
```

Question 6 *Can you figure out the definition of* ++*?*

List Concatenation: ++

++ is the operator for concatenating two lists. Look at the type:

```
*Logic> :t (++)
(++) :: forall a. [a] -> [a] -> [a]
```

Question 6 *Can you figure out the definition of* ++*?*

Answer:

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

List Concatenation: concat

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ (concat xss)
```

Question 7 *What does* `concat` *do?*

List Concatenation: concat

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ (concat xss)
```

Question 7 *What does* `concat` *do?*

Answer: `concat` takes a list of lists and constructs a single list. Example:

```
*Logic> concat [[1,2],[4,5],[7,8]]
[1,2,4,5,7,8]
```

filter and nub

Before we can explain `nub` we must understand `filter`. Here is the type:

```
filter :: (a -> Bool) -> [a] -> [a]
```

`Bool` is the type of a Boolean: True or False. So `a -> Bool` is a property.

Here is an example of how `filter` is used:

```
*Logic> filter even [2,3,5,7,8,9,10]
[2,8,10]
```

Question 8 *Can you figure out the definition of* `filter`*?*

filter and nub

Before we can explain `nub` we must understand `filter`. Here is the type:

```
filter :: (a -> Bool) -> [a] -> [a]
```

`Bool` is the type of a Boolean: True or False. So `a -> Bool` is a property.

Here is an example of how `filter` is used:

```
*Logic> filter even [2,3,5,7,8,9,10]
[2,8,10]
```

Question 8 *Can you figure out the definition of* `filter`*?*

Answer:

```
filter p [] = []
filter p (x:xs) = if p x then x : filter p xs
                         else     filter p xs
```

Removing duplicates from a list with nub

Example of the use of `nub`:

```
*Logic> nub [1,2,3,4,1,2,5]
[1,2,3,4,5]
```

Question 9 *Can you figure out the type of* `nub`*?*

Removing duplicates from a list with nub

Example of the use of `nub`:

```
*Logic> nub [1,2,3,4,1,2,5]
[1,2,3,4,5]
```

Question 9 *Can you figure out the type of* `nub`*?*

Answer: `nub :: Eq a => [a] -> [a]`. The `Eq a` means that equality has to be defined for `a`.

Question 10 *Can you figure out the definition of* `nub`*?*

Removing duplicates from a list with nub

Example of the use of `nub`:

```
*Logic> nub [1,2,3,4,1,2,5]
[1,2,3,4,5]
```

Question 9 *Can you figure out the type of* `nub`*?*

Answer: `nub :: Eq a => [a] -> [a]`. The `Eq a` means that equality has to be defined for `a`.

Question 10 *Can you figure out the definition of* `nub`*?*

Answer:

```
nub [] = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

Sorting a list

In order to sort a list (put their elements in some order), we need to be able to compare their elements for size. This is can be done with `compare`:

```
*Logic> compare 3 4
LT
*Logic> compare 'C' 'B'
GT
*Logic> compare [3] [3]
EQ
*Logic> compare "smart" "smile"
LT
```

In order to define our own sorting algorithm, let's first define a function for inserting an item at the correct position in an ordered list.

Question 11 *Can you figure out how to do that? The type is*

```
myinsert :: Ord a => a -> [a] -> [a].
```

Sorting a list

In order to sort a list (put their elements in some order), we need to be able to compare their elements for size. This is can be done with `compare`:

```
*Logic> compare 3 4
LT
*Logic> compare 'C' 'B'
GT
*Logic> compare [3] [3]
EQ
*Logic> compare "smart" "smile"
LT
```

In order to define our own sorting algorithm, let's first define a function for inserting an item at the correct position in an ordered list.

Question 11 *Can you figure out how to do that? The type is*

```
myinsert :: Ord a => a -> [a] -> [a].
```

Answer:

```
myinsert :: Ord a => a -> [a] -> [a]
myinsert x [] = [x]
myinsert x (y:ys) = if compare x y == GT
                      then y : myinsert x ys
                      else x:y:ys
```

Answer:

```
myinsert :: Ord a => a -> [a] -> [a]
myinsert x [] = [x]
myinsert x (y:ys) = if compare x y == GT
                      then y : myinsert x ys
                      else x:y:ys
```

Question 12 *Can you now implement your own sorting algorithm? The type is*

`mysort :: Ord a => [a] -> [a].`

Answer:

```
myinsert :: Ord a => a -> [a] -> [a]
myinsert x [] = [x]
myinsert x (y:ys) = if compare x y == GT
                       then y : myinsert x ys
                       else x:y:ys
```

Question 12 *Can you now implement your own sorting algorithm? The type is* `mysort :: Ord a => [a] -> [a].`

```
mysort :: Ord a => [a] -> [a]
mysort [] = []
mysort (x:xs) = myinsert x (mysort xs)
```

Valuations

```
type Valuation = [(Name,Bool)]
```

All possible valuations for list of prop letters:

```
genVals :: [Name] -> [Valuation]
genVals [] = [[]]
genVals (name:names) =
  map ((name,True) :) (genVals names)
  ++ map ((name,False):) (genVals names)
```

All possible valuations for a formula, with function composition:

```
allVals :: Form -> [Valuation]
allVals = genVals . propNames
```

Composing functions with ‘.’

The composition of two functions f and g, pronounced ‘f after g’ is the function that results from first applying g and next f.

Standard notation for this: $f \cdot g$. This is pronounced as “f after g”.

Haskell implementation:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
f . g = \ x -> f (g x)
```

Note the types! Note the lambda abstraction.

Lambda Abstraction

In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

The standard mathematical notation for this is $\lambda x \mapsto x * x$. Haskell notation aims at remaining close to mathematical notation.

- The intention is that variabele `x` stands proxy for a number of type `Int`.
- The result, the squared number, also has type `Int`.
- The function `sqr` is a function that, when combined with an argument of type `Int`, yields a value of type `Int`.
- This is precisely what the type-indication `Int -> Int` expresses.

Blowup

Question 13 *If a propositional formula has 20 variables, how many different valuations are there for that formula?*

Blowup

Question 13 *If a propositional formula has 20 variables, how many different valuations are there for that formula?*

Answer: look at this:

```
*Logic> map (2^) [1..20]
[2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,
 16384,32768,65536,131072,262144,524288,1048576]
```

The number doubles with every extra variable, so it grows exponentially.

Question 14 *Does the definition of* `genVals` *use a feasible algorithm?*

Blowup

Question 13 *If a propositional formula has 20 variables, how many different valuations are there for that formula?*

Answer: look at this:

```
*Logic> map (2^) [1..20]
[2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,
 16384,32768,65536,131072,262144,524288,1048576]
```

The number doubles with every extra variable, so it grows exponentially.

Question 14 *Does the definition of* `genVals` *use a feasible algorithm?*

Answer: no.

Evaluation of Formulas

```
eval :: Valuation -> Form -> Bool
eval [] (Prop c)     = error ("no info: " ++ show c)
eval ((i,b):xs) (Prop c)
     | c == i     = b
     | otherwise = eval xs (Prop c)
eval xs (Neg f)   = not (eval xs f)
eval xs (Cnj fs) = all (eval xs) fs
eval xs (Dsj fs) = any (eval xs) fs
eval xs (Impl f1 f2) =
     not (eval xs f1) || eval xs f2
eval xs (Equiv f1 f2) = eval xs f1 == eval xs f2
```

Question 15 *Does the definition of* `eval` *use a feasible algorithm?*

Evaluation of Formulas

```
eval :: Valuation -> Form -> Bool
eval [] (Prop c)     = error ("no info: " ++ show c)
eval ((i,b):xs) (Prop c)
     | c == i     = b
     | otherwise = eval xs (Prop c)
eval xs (Neg f)   = not (eval xs f)
eval xs (Cnj fs) = all (eval xs) fs
eval xs (Dsj fs) = any (eval xs) fs
eval xs (Impl f1 f2) =
     not (eval xs f1) || eval xs f2
eval xs (Equiv f1 f2) = eval xs f1 == eval xs f2
```

Question 15 *Does the definition of* `eval` *use a feasible algorithm?*

Answer: yes.

New functions: not, all, any

Let’s give our own implementations:

New functions: not, all, any

Let’s give our own implementations:

```
mynot :: Bool -> Bool
mynot True = False
mynot False = True
```

New functions: not, all, any

Let’s give our own implementations:

```
mynot :: Bool -> Bool
mynot True = False
mynot False = True
```

```
myall :: Eq a => (a -> Bool) -> [a] -> Bool
myall p [] = True
myall p (x:xs) = p x && myall p xs
```

New functions: not, all, any

Let’s give our own implementations:

```
mynot :: Bool -> Bool
mynot True = False
mynot False = True
```

```
myall :: Eq a => (a -> Bool) -> [a] -> Bool
myall p [] = True
myall p (x:xs) = p x && myall p xs
```

```
myany :: Eq a => (a -> Bool) -> [a] -> Bool
myany p [] = False
myany p (x:xs) = p x || myany p xs
```

Satisfiability

A formula is satisfiable if some valuation makes it true.

We know what the valuations of a formula `f` are. These are given by

```
allVals f
```

We also know how to express that a valuation `v` makes a formula `f` true:

```
eval v f
```

This gives:

```
satisfiable :: Form -> Bool
satisfiable f = any (\ v -> eval v f) (allVals f)
```

Some hard questions

Question 16 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

Some hard questions

Question 16 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

Answer: no, for the call to `allVals` causes an exponential blowup.

Question 17 *Can you think of a feasible algorithm for* `satisfiable`*?*

Some hard questions

Question 16 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

Answer: no, for the call to `allVals` causes an exponential blowup.

Question 17 *Can you think of a feasible algorithm for* `satisfiable`*?*

Answer: not very likely . . .

Question 18 *Does a feasible algorithm for* `satisfiable` *exist?*

Some hard questions

Question 16 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

Answer: no, for the call to `allVals` causes an exponential blowup.

Question 17 *Can you think of a feasible algorithm for* `satisfiable`*?*

Answer: not very likely . . .

Question 18 *Does a feasible algorithm for* `satisfiable` *exist?*

Answer: nobody knows. This is the famous P versus NP problem. If I am allowed to guess a valuation for a formula, then the `eval` check whether the valuation makes the formula true takes a polynomial number of steps in the size of the formula. But I first have to find such a valuation, and the number of candidates is exponential in the size of the formula. All known algorithms for `satisfiable` take an exponential number of steps, in the worst case . . .

Let’s Talk a Bit About Syntax

Let’s Talk a Bit About Syntax

Noam Chomsky (born 1928)

Lexical Scanning

Tokens:

```
data Token
      = TokenNeg
      | TokenCnj
      | TokenDsj
      | TokenImpl
      | TokenEquiv
      | TokenInt Int
      | TokenOP
      | TokenCP
 deriving (Show,Eq)
```

The lexer converts a string to a list of tokens.

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs) | isSpace c = lexer cs
             | isDigit c = lexNum (c:cs)
lexer ('(':cs) = TokenOP : lexer cs
lexer (')':cs) = TokenCP : lexer cs
lexer ('*':cs) = TokenCnj : lexer cs
lexer ('+':cs) = TokenDsj : lexer cs
lexer ('-':cs) = TokenNeg : lexer cs
lexer ('=':'=':'>':cs) = TokenImpl : lexer cs
lexer ('<':'=':'>':cs) = TokenEquiv : lexer cs
lexer (x:_) = error ("unknown token: " ++ [x])
```

```
*Logic> lexer "*(2 3 -4 +("
[TokenCnj,TokenOP,TokenInt 2,TokenInt 3,TokenNeg,
 TokenInt 4,TokenDsj,TokenOP]
```

Parsing

A parser for token type `a` that constructs a datatype `b` has the following type:

```
type Parser a b = [a] -> [(b,[a])]
```

The parser constructs a list of tuples `(b,[a])` from an initial segment of a token string `[a]`. The remainder list in the second element of the result is the list of tokens that were not used in the construction of the datatype.

If the output list is empty, the parse has not succeeded. If the output list more than one element, the token list was ambiguous.

Success

The parser that succeeds immediately, while consuming no input:

```
succeed :: b -> Parser a b
succeed x xs = [(x,xs)]
```

```
parseForm :: Parser Token Form
parseForm (TokenInt x: tokens) = [(Prop x,tokens)]
parseForm (TokenNeg : tokens) =
  [ (Neg f, rest) | (f,rest) <- parseForm tokens ]
parseForm (TokenCnj : TokenOP : tokens) =
  [ (Cnj fs, rest) | (fs,rest) <- parseForms tokens ]
parseForm (TokenDsj : TokenOP : tokens) =
  [ (Dsj fs, rest) | (fs,rest) <- parseForms tokens ]
parseForm (TokenOP : tokens) =
  [ (Impl f1 f2, rest) | (f1,ys) <- parseForm tokens,
                         (f2,rest) <- parseImpl ys ]
   ++
  [ (Equiv f1 f2, rest) | (f1,ys) <- parseForm tokens,
                          (f2,rest) <- parseEquiv ys ]
parseForm tokens = []
```

List Comprehension

List comprehension is defining lists by the following method:

```
[ x | x <- xs, property x ]
```

This defines the sublist of `xs` of all items satisfying `property`. It is equivalent to:

```
filter property xs
```

Example:

```
someEvens     = [ x | x <- [1..1000], even x ]
```

Equivalently:

```
someEvens     = filter even [1..1000]
```

Parsing a list of formulas

Success if a closing parenthesis is encountered.

```
parseForms :: Parser Token [Form]
parseForms (TokenCP : tokens) = succeed [] tokens
parseForms tokens =
   [(f:fs, rest) | (f,ys) <- parseForm tokens,
                   (fs,rest) <- parseForms ys ]
```

The Parse Function

```
parse :: String -> [Form]
parse s = [ f | (f,_) <- parseForm (lexer s) ]
```

```
*Logic> parse "*(1 +(2 -3))"
[*(1 +(2 -3))]
*Logic> parse "*(1 +(2 -3)"
[]
*Logic> parse "*(1 +(2 -3))))"
[*(1 +(2 -3))]
*Logic> parseForm (lexer "*(1 +(2 -3))))")
[(*(1 +(2 -3)),[TokenCP,TokenCP])]
```

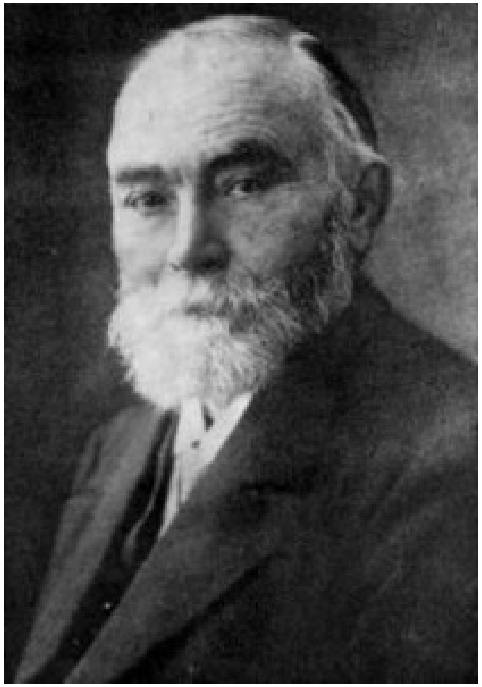
Further Exercises

You are invited to write implementations of:

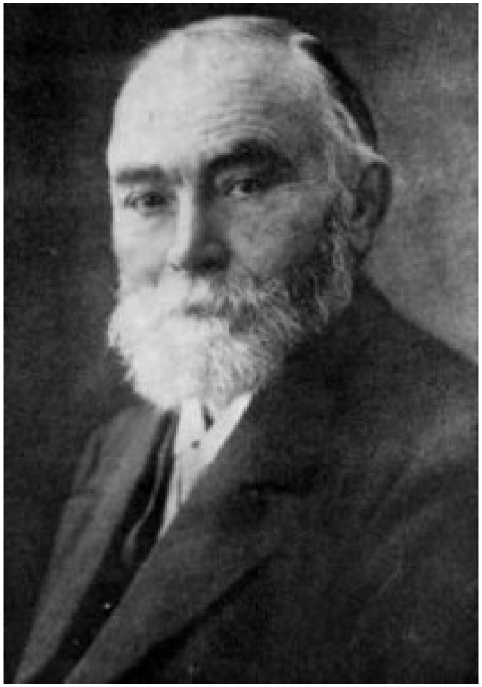
```
contradiction :: Form -> Bool

tautology :: Form -> Bool

-- logical entailment
entails :: Form -> Form -> Bool

-- logical equivalence
equiv :: Form -> Form -> Bool
```

and to test your definitions for correctness.

Who is Who in Logic?



Who is Who in Logic?



Gottlob Frege (1848 – 1925)

Predicate Logic

Assume a set of function symbols is given, and let f range over function symbols.

Assume a set of predicate symbols is given, and let P range over predicate symbols.

$$\begin{array}{rcl} t & ::= & x \mid f(t_1, \ldots, t_n) \\ \varphi & ::= & P(t_1, \ldots, t_n) \mid t_1 = t_2 \\ & & \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (\varphi \leftrightarrow \varphi) \\ & & \mid (\forall x \varphi) \mid (\exists x \varphi) \end{array}$$

Syntax of First Order Logic in Haskell: Terms

```haskell
type Nm = String
data Term = V Nm | F Nm [Term] deriving (Eq,Ord)
```

```haskell
x, y, z :: Term
x = V "x"
y = V "y"
z = V "z"
```

Operations on Terms: Finding the Variables

```
varsInTerm :: Term -> [Nm]
varsInTerm (V name) = [name]
varsInTerm (F _ ts) = varsInTerms ts where
  varsInTerms :: [Term] -> [Nm]
  varsInTerms = nub . concat . map varsInTerm
```

This is similar to finding the names in a propositional formula.

Syntax of First Order Logic in Haskell: Formulas

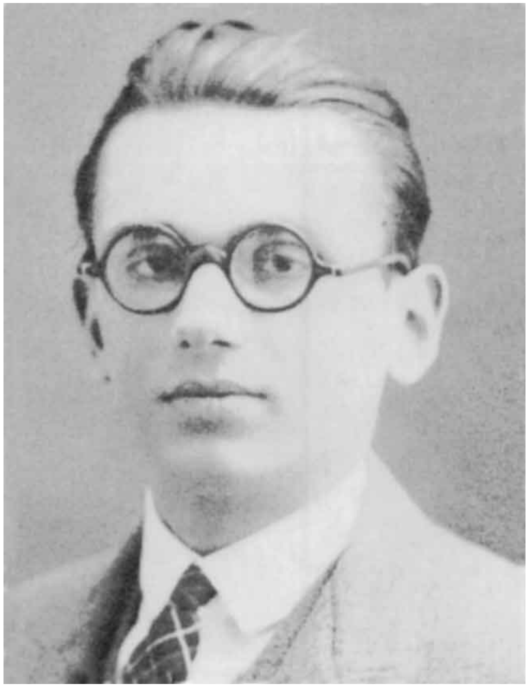
```
data Formula = Atom Nm [Term]
               | Eq Term Term
               | Ng  Formula
               | Imp Formula Formula
               | Equi Formula Formula
               | Conj [Formula]
               | Disj [Formula]
               | Forall Nm Formula
               | Exists Nm Formula
               deriving (Eq,Ord)
```

```
r0 = Atom "R"

formula1 = Forall "x" (r0 [x,x])
formula2 = Forall "x"
             (Forall "y"
               (Imp (r0 [x,y]) (r0 [y,x])))
formula3 = Forall "x"
             (Forall "y"
              (Forall "z"
               (Imp (Conj [r0 [x,y], r0 [y,z]])
                    (r0 [x,z])))))
```

```
r0 = Atom "R"

formula1 = Forall "x" (r0 [x,x])
formula2 = Forall "x"
             (Forall "y"
               (Imp (r0 [x,y]) (r0 [y,x])))
formula3 = Forall "x"
             (Forall "y"
              (Forall "z"
               (Imp (Conj [r0 [x,y], r0 [y,z]])
                    (r0 [x,z])))))
```

```
*Logic> formula1
A x R[x,x]
*Logic> formula2
A x A y (R[x,y]==>R[y,x])
*Logic> formula3
A x A y A z (*[R[x,y],R[y,z]]==>R[x,z])
```

Who is Who in Logic (2)?



Who is Who in Logic (2)?

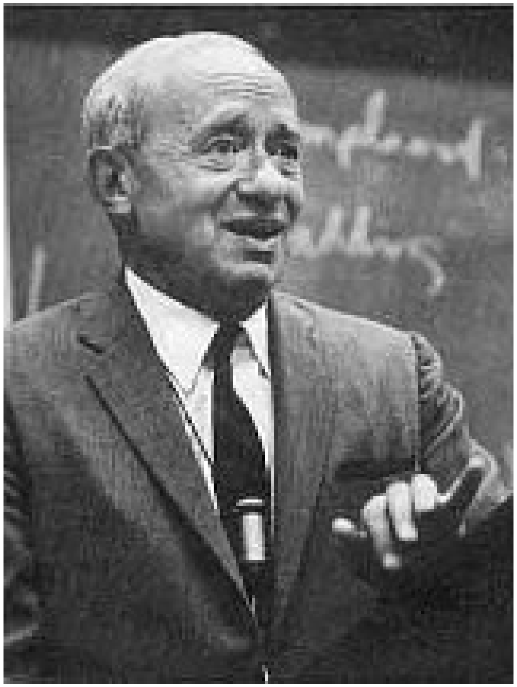
Kurt Gödel (1906 – 1978)

Semi-decidability of First Order Predicate Logic

Kurt Gödel showed in his PhD thesis (1929) that a sound and complete proof calculus for first order predicate logic exists.

It follows from this that the notion of logical consequence for predicate logic is semi-decidable. Since proofs are finite, it is possible to enumerate all finite derivations, so there exists a finite enumeration of pairs of first order sentences (φ, ψ) such that ψ is a logical consequence of φ.

Who is Who in Logic (3)?



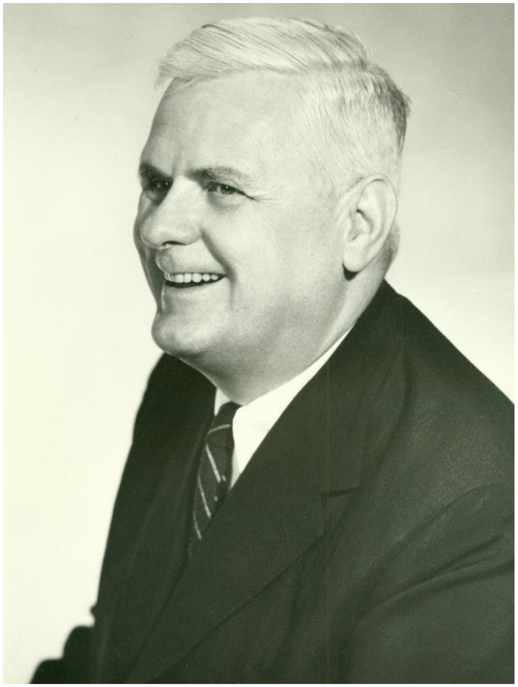
Who is Who in Logic (3)?

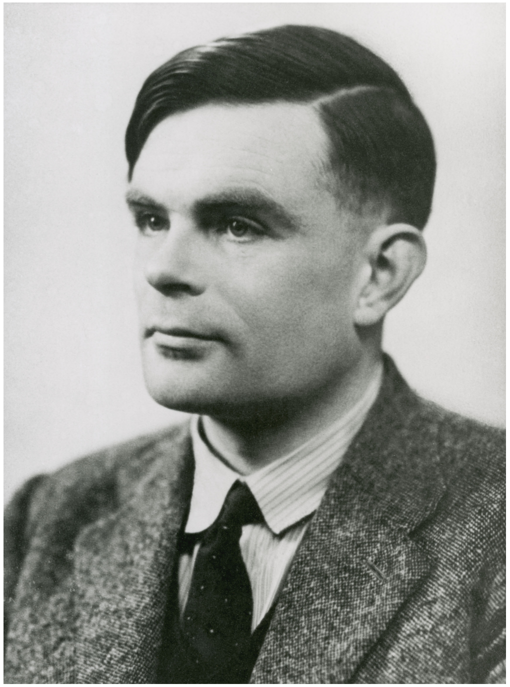
Alfred Tarski (1901 – 1983)

Notion of Truth in Formal Languages

```
evalFOL :: Eq a =>
    [a] -> Lookup a -> Fint a -> Rint a -> Formula -> Bool
evalFOL domain g f i = evalFOL' g where
  evalFOL' g (Atom name ts) = i name (map (termVal g f) ts)
  evalFOL' g (Eq t1 t2) = termVal g f t1 == termVal g f t2
  evalFOL' g (Ng form) = not (evalFOL' g form)
  evalFOL' g (Imp f1 f2) = not
                    (evalFOL' g f1 && not (evalFOL' g f2))
  evalFOL' g (Equi f1 f2) = evalFOL' g f1 == evalFOL' g f2
  evalFOL' g (Conj fs)    = and (map (evalFOL' g) fs)
  evalFOL' g (Disj fs)    = or  (map (evalFOL' g) fs)
  evalFOL' g (Forall v form) =
    all (\ d -> evalFOL' (changeLookup g v d) form) domain
  evalFOL' g (Exists v form) =
    any (\ d -> evalFOL' (changeLookup g v d) form) domain
```

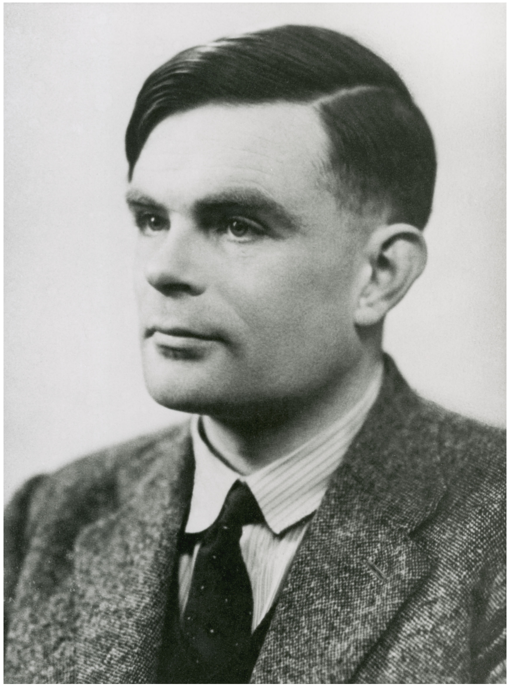
Who is Who in Logic (4)?





Who is Who in Logic (4)?

Alonzo Church (1903 – 1995)


Alan Turing (1912 – 1954)

Limitations of First Order Predicate Logic

CLAUDE PRESBYTERIAN CHURCH
THERE ARE SOME QUESTIONS THAT CAN'T BE ANSWERED BY GOOGLE
Citytv
CityNews
CityNews.ca

Some New Billboards

There are some questions
that can’t be answered by logic

There are some questions
that can’t be answered
by computing machines

Undecidable Queries

Undecidable Queries

- The deep reason behind the undecidability of first order logic is the fact that its expressive power is so great that it is possible to state undecidable queries.

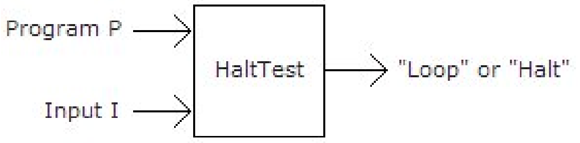
Undecidable Queries

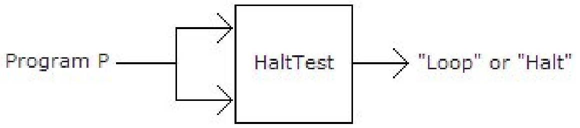
- The deep reason behind the undecidability of first order logic is the fact that its expressive power is so great that it is possible to state undecidable queries.
- One of the famous undecidable queries is the halting problem.

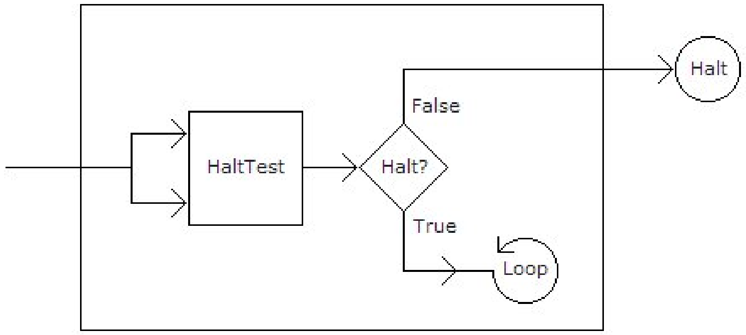
Undecidable Queries

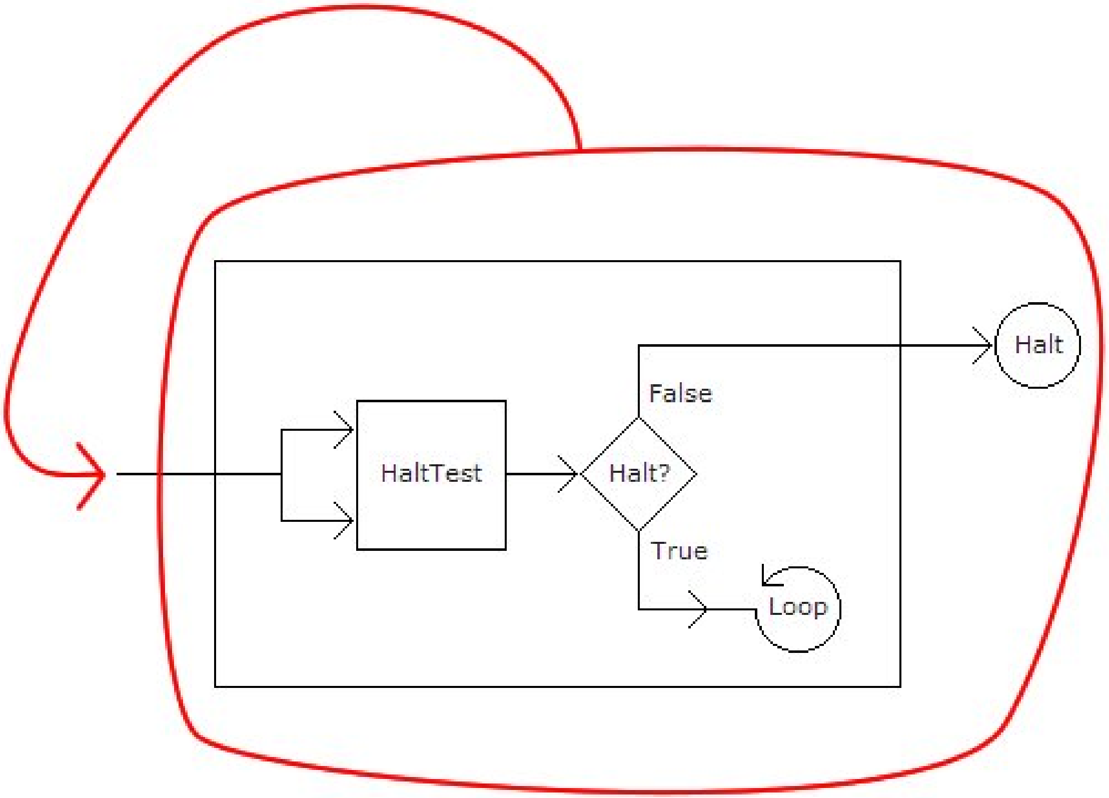
- The deep reason behind the undecidability of first order logic is the fact that its expressive power is so great that it is possible to state undecidable queries.
- One of the famous undecidable queries is the halting problem.
- Here is what a halting algorithm would look like:
  - Input: a specification of a computational procedure P, and an input I for that procedure
  - Output: an answer 'halt' if P halts when applied to I, and 'loop' otherwise.

Undecidability of the Halting Problem, in pictures …





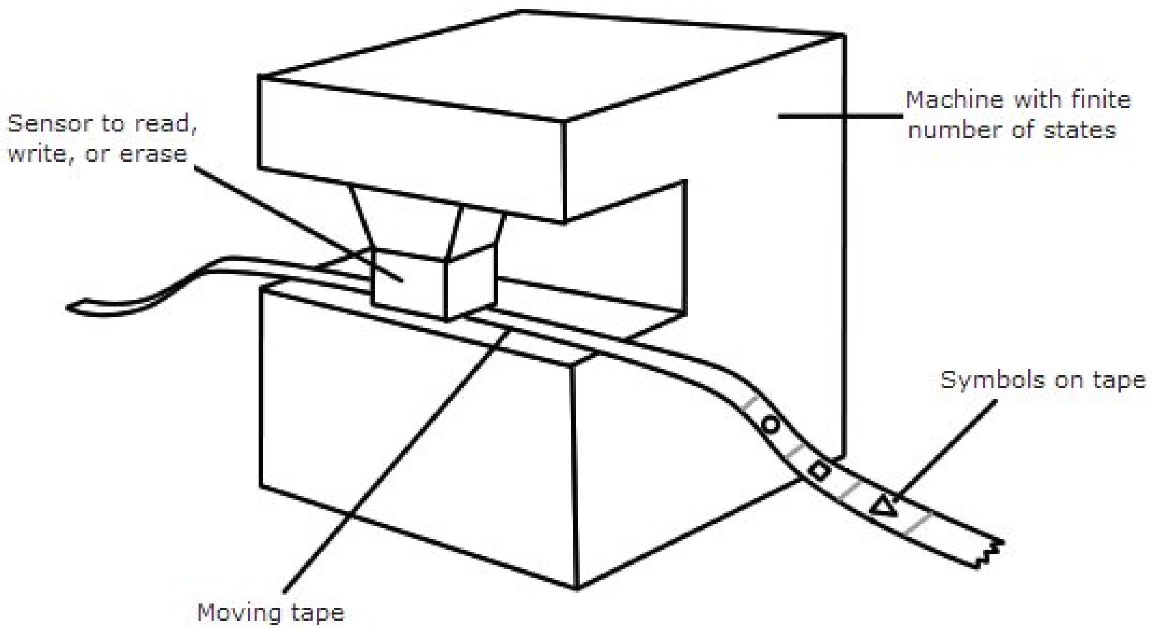



HaltTest
Halt?
False
True
Halt
Loop

Alan Turing's Insight

- A language that allows the specification of 'universal procedures' such as HaltTest cannot be decidable.
- But first order predicate logic is such a language . . .
- The formal proof of the undecidability of first order logic consists of
  - A **very general** definition of computational procedures.
  - A demonstration of the fact that such computational procedures can be expressed in first order logic.
  - A demonstration of the fact that the halting problem for computational procedures is undecidable (see the picture above).
  - A formulation of the halting problem in first order logic.
- This formal proof was provided by Alan Turing in [Tur36]. The computational procedures he defined for this purpose were later called Turing machines.

Picture of a Turing Machine


Sensor to read, write, or erase
Machine with finite number of states
Symbols on tape
Moving tape

A Turing Machine

Application of Logic in Programming: Formal Specification

An invariant of a function F in a state s is a condition C with the property that if C holds in s then C will also hold in any state that results from execution of F in s.

Thus, invariants are assertions of the type:

```
invar1 :: (a -> Bool) -> (a -> a) -> a -> a
invar1 p f x =
  let
    x' = f x
  in
  if p x && not (p x') then error "invar1"
  else x'
```

Example: every natural number N can be written in the form $N = 2^k * m$ where m is odd. Here is the algorithm:

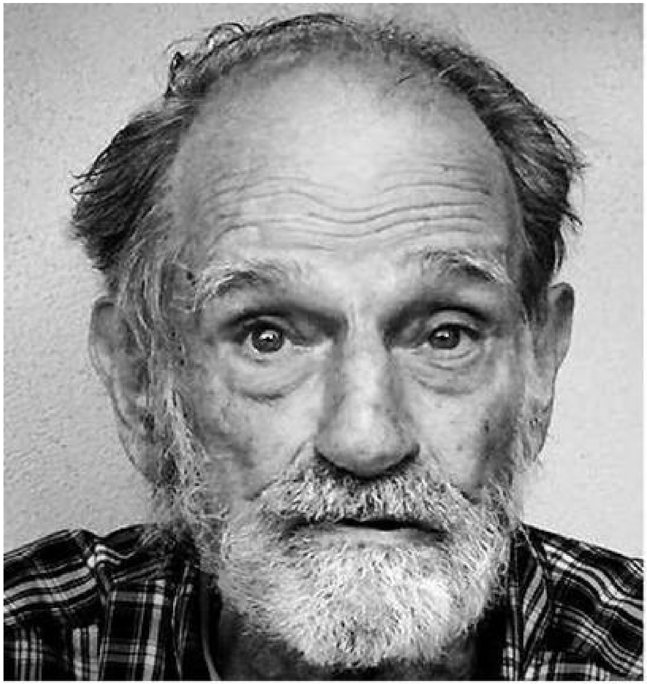
```
decomp :: Integer -> (Integer,Integer)
decomp n = decomp' (0,n)

decomp' :: (Integer,Integer) -> (Integer,Integer)
decomp' = until (\ (_,m) -> odd m)
                (\ (k,m) -> (k+1,div m 2))
```

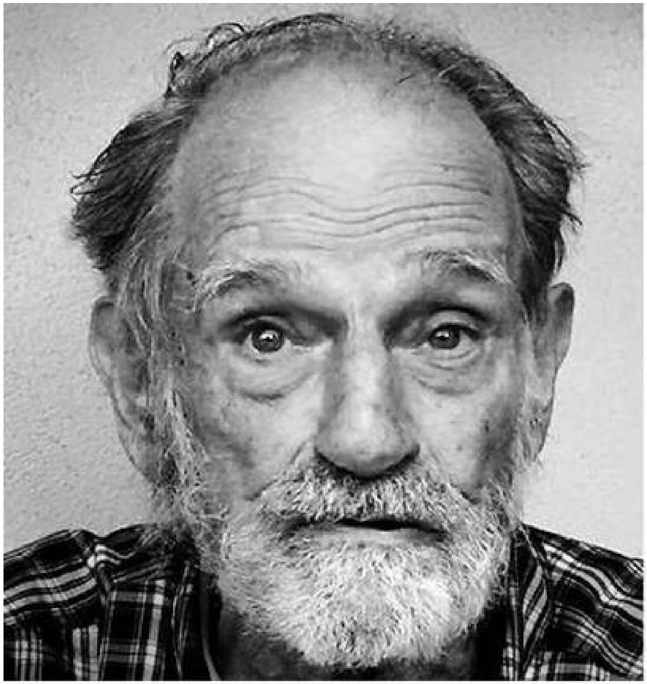
And here is the invariant for the algorithm:

```
invarDecomp :: Integer -> (Integer,Integer)
                       -> (Integer,Integer)
invarDecomp n = invar1
                (\ (i,j) -> 2^i*j == n)
                decomp'
```

Stability of Relations

Question 19 *For people who are with a partner: how can you find out if your relationship is stable?*





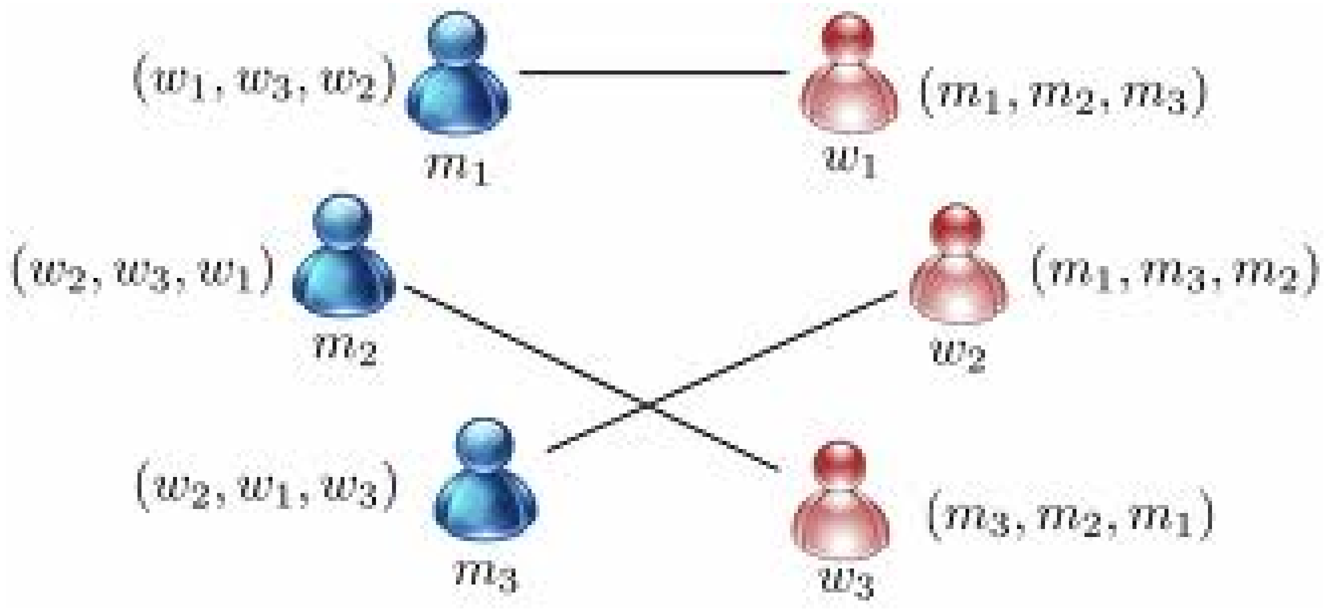
Lloyd Shapley (born 1923)

The Stable Marriage Problem

Suppose equal sets of men and women are given, each man has listed the women in order of preference, and vice versa for the women.

A stable marriage match between men and women is a one-to-one mapping between the men and women with the property that if a man prefers another woman over his own wife then that woman does not prefer him to her own husband, and if a woman prefers another man over her own husband, then that man does not prefer her to his own wife.

Example



Gale and Shapley: Stable Matchings Always Exist

The mathematicians/economists Gale and Shapley proved that stable matchings always exist, and gave an algorithm for finding such matchings, the so-called Gale-Shapley algorithm [GS62]. This has many important applications, also outside of the area of marriage counseling.

See http://en.wikipedia.org/wiki/Stable_marriage_problem for more information.

In 2012 Lloyd S. Shapley received the Nobel prize in Economics. Prize motivation: "For the theory of stable allocations and the practice of market design." (David Gale died in 2008.)

http://www.nobelprize.org/nobel_prizes/economic-sciences/laureates/2012/shapley-diploma.html

Gale-Shapley

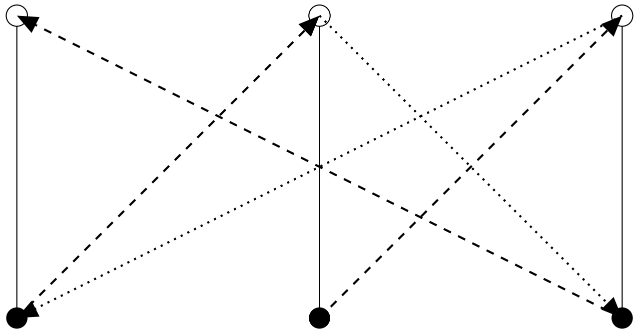
Gale-Shapley algorithm for stable marriage

```
1. Initialize all $m$ in $M$ and $w$ in $W$ to free;
2. while there is a free man $m$ who still has a woman $w$ to propose to
   (a) $w$ is the highest ranked woman to whom $m$ has not yet proposed
   (b) if $w$ is free, $(w, m)$ become engaged
       else (some pair $(w, m')$ already exists)
       if $w$ prefers $m$ to $m'$
          i. $(w, m)$ become engaged
         ii. $m'$ becomes free
       else $(w, m')$ remain engaged
```

Stable Match?

```
mt  =  [(1,[2,1,3]),  (2,  [3,2,1]),  (3,[1,3,2])]
wt  =  [(1,[1,2,3]),(2,[3,2,1]),  (3,[1,3,2])]
```

○ for the women, ● for the men:



Stability Property

An invariant of the Gale-Shapley algorithm is the stability property. Here is the definition of stability for a relation E consisting of engaged (w, m) pairs:

$$\forall (w, m) \in E \forall (w', m') \in E$$
$$((\mathrm{pr}_w m' m \to \mathrm{pr}_{m'} w' w) \wedge (\mathrm{pr}_m w' w \to \mathrm{pr}_{w'} m' m)).$$

What this says is: if w prefers another guy m' to her own fiancee m then m' does prefer his own fiancee w' to w, and if m prefers another woman w' to his own fiancee w then w' does prefer her own fiancee m' to m.

Haskell version, useful for model checking the outcome of the algorithm:

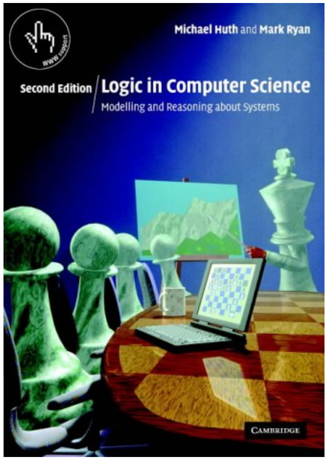
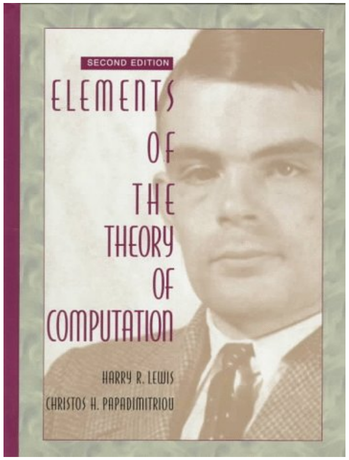
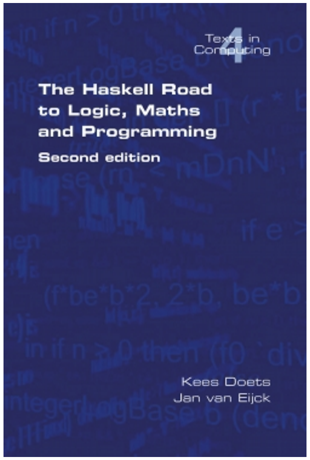
```
isStable :: Wpref -> Mpref -> Engaged -> Bool
isStable wp mp engaged =
    forall engaged (\ (w,m) -> forall engaged
          (\ (w',m') -> (wp w m' m ==> mp m' w' w)
                         &&
                        (mp m w' w ==> wp w' m' m)))
```

Links and Books for Further Study

http://www.logicinaction.org

http://www.cwi.nl/~jve/HR

http://projecteuler.net


Michael Huth and Mark Ryan
Second Edition
Logic in Computer Science
Modelling and Reasoning about Systems
Cambridge

SECOND EDITION
ELEMENTS OF THE THEORY OF COMPUTATION
HARRY R. LEWIS
CHRISTOS H. PAPADIMITRIOU


References

[GS62] D. Gale and L. Shapley. College admissions and the stability of marriage. American Mathematical Monthly, 69:9–15, 1962.

[Tur36] A.M. Turing. On computable real numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 2(42):230–265, 1936.