Logic, (Functional) Programming, Model Checking

Jan van Eijck
CWI & ILLC, Amsterdam

Tsinghua University, November 2, 2015

Abstract

This lecture will set the stage by combining the topics of the title in various ways. First I will remind you that logic is part of every programming language, in the form of boolean expressions. Next, we will do a computational analysis of the language of boolean expressions, looking both at syntax and semantics. If the language of boolean expressions is enriched with quantifiers, we move from propositional logic to predicate logic. We will discuss how the expressions of that language can describe the ways things are, and we will look at model checking, and at the reverse side of the expressive power of predicate logic. The lecture ends with a brief sketch of epistemic model checking. In the course of the lecture we 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 condition is true then do this, else do that.

Loop As long as condition is true, do this.

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

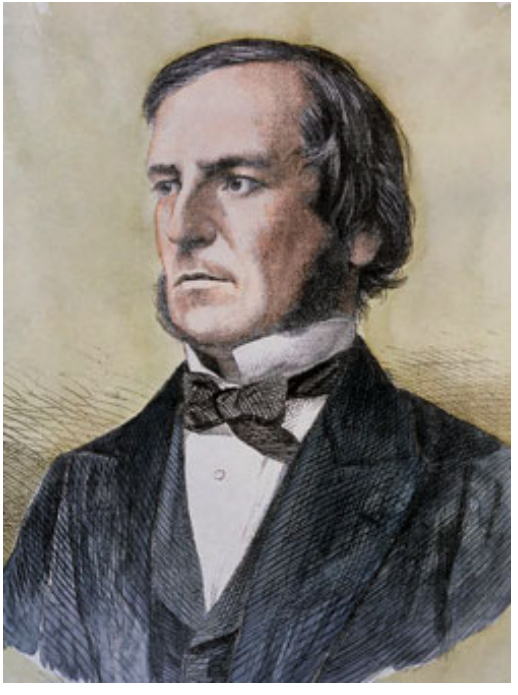
Let's Talk About the Logic

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 LFPMC

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 0
q = Prop 1
r = Prop 2
s = 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 Elements

Haskell has a predefined function `elem :: Eq a => a -> [a] -> Bool`.
Let's figure out how this works by giving our own definition.

List Elements

Haskell has a predefined function `elem :: Eq a => a -> [a] -> Bool`. Let's figure out how this works by giving our own definition.

```
myElem :: Eq a => a -> [a] -> Bool
myElem x [] = False
myElem x (y:ys) = x==y || myElem x 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]
o[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]
o[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 *Can you figure out the definitions of* `all` *and* `any`*?*

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 *Can you figure out the definitions of* `all` *and* `any`*?*

Let's do our own implementations:

```
myall p = and . map p
myany p = or . map p
```

Question 16 *As you can see, these are definitions in terms of* `and` *and* `or`*. Can we define these as well?*

```
myall p = and . map p
myany p = or . map p
```

Question 16 *As you can see, these are definitions in terms of* `and` *and* `or`*. Can we define these as well?*

```
myAnd [] = True
myAnd (b:bs) = b && myAnd bs

myOr [] = False
myOr (b:bs) = b || myOr bs
```

Question 17 *Does the definition of* `eval` *use a feasible algorithm?*

```
myall p = and . map p
myany p = or . map p
```

Question 16 *As you can see, these are definitions in terms of* `and` *and* `or`*. Can we define these as well?*

```
myAnd [] = True
myAnd (b:bs) = b && myAnd bs

myOr [] = False
myOr (b:bs) = b || myOr bs
```

Question 17 *Does the definition of* `eval` *use a feasible algorithm?*

Answer: yes.

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

Some hard questions

Question 18 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

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

Question 19 *Can you think of a feasible algorithm for* `satisfiable`*?*

Some hard questions

Question 18 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

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

Question 19 *Can you think of a feasible algorithm for* `satisfiable`*?*

Answer: not very likely . . .

Question 20 *Does a feasible algorithm for* `satisfiable` *exist?*

Some hard questions

Question 18 *Is the algorithm used in the definition of* `satisfiable` *a feasible algorithm?*

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

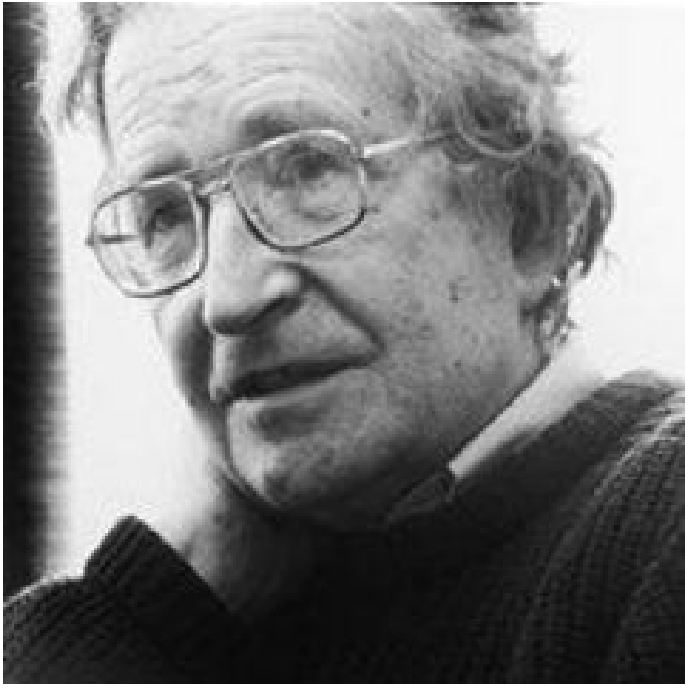
Question 19 *Can you think of a feasible algorithm for* `satisfiable`*?*

Answer: not very likely . . .

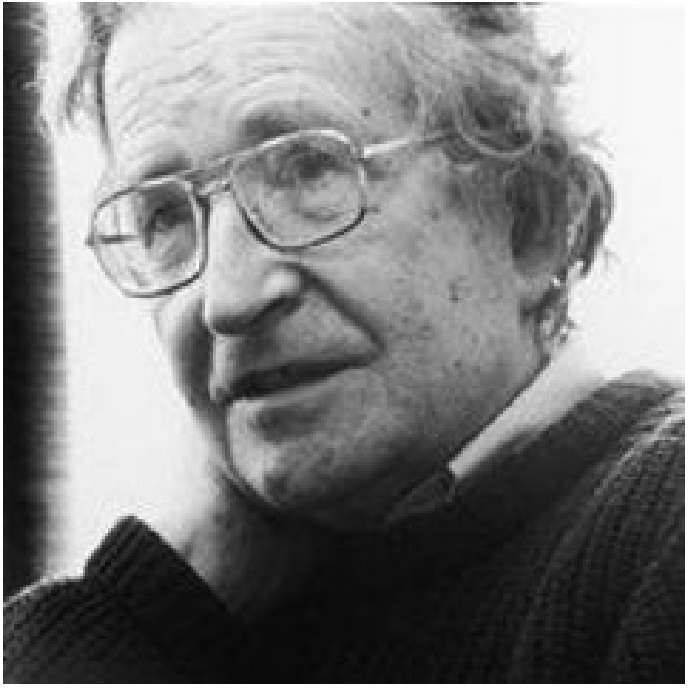
Question 20 *Does a feasible algorithm for* `satisfiable` *exist?*

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.

Epistemic Model Checking with Propositional Logic

In four easy steps . . .

Epistemic Model Checking with Propositional Logic

In four easy steps . . .

Step 1. Determine your Universe.

```
universe :: Int -> [Valuation]
universe n = genVals [0..n-1]
```

Epistemic Model Checking with Propositional Logic

In four easy steps . . .

Step 1. Determine your Universe.

```
universe :: Int -> [Valuation]
universe n = genVals [0..n-1]
```

Step 2. Determine your update function.

```
update :: Form -> [Valuation] -> [Valuation]
update f = filter (\v -> eval v f)
```

Step 3. Extend the update function to lists of formulas.

```
updates :: [Form] -> [Valuation] -> [Valuation]
updates [] vs = vs
updates (f:fs) vs = updates fs (update f vs)
```

Step 3. Extend the update function to lists of formulas.

```
updates :: [Form] -> [Valuation] -> [Valuation]
updates [] vs = vs
updates (f:fs) vs = updates fs (update f vs)
```

Step 4: Model-check in the result state.

```
check :: [Valuation] -> Form -> Bool
check vs f = all (\v -> eval v f) vs
```

Example

```
fs = [Impl p q, Impl q r, Impl r s]
f  = Impl p s
```

```
exampleSpace = updates fs (universe 4)
```

```
exampleCheck = check exampleSpace f
```

Question 21 *Can you predict the outcome?*

Further Exercises

See http://homepages.cwi.nl/~jve/courses/15/tsinghua/Exercises1.html