Computational Semantics

With

Functional Programming

Solutions to the Exercises

— December 4, 2009—

Jan van Eijck and Christina Unger
Solutions to Exercises from Chapter 1

1.1 For 10 facts there are \(2^{10} = 1024\) possibilities. In general, \(n\) facts yield \(2^n\) possibilities. Every additional fact doubles the set of possibilities. The expressiveness of propositional logic is exponential in the number of atomic facts.

1.2 Using \((X)^*\) for an arbitrary finite number of copies of \(X\), we can characterize the pattern as: “Sentences can go on (and on)*.”

1.3 It does follow from the example that there are infinitely many sentences. Still, each sentence has finite length. Finite sets of infinite things are different from infinite sets of finite things. The set of sentences of English is an example of an infinite set of finite things. So ‘Sentences can go on and on’ does not mean that a single sentence can go on and on, but rather that the process of building longer and longer sentences can go on and on.

Solutions to Exercises from Chapter 2

2.1 Since \(\emptyset\) has no members, it holds trivially that every member of \(\emptyset\) is a member of \(A\), i.e. that \(\emptyset \subseteq A\).

2.2 The empty set \(\emptyset\) has no members, whereas \(\{\emptyset\}\) has exactly one member, namely \(\emptyset\).

2.3
\[
\overline{A} = U - A = U - (U - A) = \{x \in U \mid x \notin (U - A)\} = \{x \in U \mid x \in A\} = A.
\]

2.4
\[
\{(Ks, Ks), (Ks, Kr), (Ks, A), (Kr, Ks), (Kr, Kr), (Kr, A), (A, Ks), (A, Kr), (A, A)\}.
\]

2.5 \(\{(n, n + 4) \mid n \in \mathbb{N}\}\).

2.6 Since \(R' \subseteq R\) is given, we only have to show the other half of \(R = R'\), i.e. we have to show that \(R \subseteq R'\). For that, assume \((x, y) \in R\). Then \((y, x) \in R'\). It
follows from this, by $R' \subseteq R$, that $(y, x) \in R$. But this means that $(x, y) \in R'$. Since we have shown that if an arbitrary pair $(x, y)$ is in $R$ it is also in $R'$, we have shown that $R \subseteq R'$.

2.7

(1) not transitive, because the relation contains $(1, 2)$ and $(2, 3)$ but not $(1, 3)$

(2) not transitive, because the relation contains $(1, 3)$ and $(3, 4)$ but not $(1, 4)$

(3) transitive

(4) not transitive, because the relation contains $(1, 2)$ and $(2, 1)$ but not $(1, 1)$

(5) transitive

2.8 This follows from the fact that the condition ‘for all $x, y, z$ it holds that if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R’” is equivalent to: ‘for all $x, z$ it holds that if there is a $y$ with $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R’”.

2.9. An example is the relation ‘less than’ on the natural numbers. It is transitive, but if we compose it with itself, we get the relation \{$(n, n + 2)$ | $n \in \mathbb{N}$\} (‘at least two less than’), and this is not the same as ‘less than’.

2.10 If $s$ is given by $n \mapsto n + 1$, then $s \cdot s$ is given by $n \mapsto n + 2$.

2.11. The characteristic function that corresponds with the relation $\leq$ on the natural numbers is the function $f: (\mathbb{N}, \mathbb{N}) \to \{\text{True, False}\}$ given by $f(n, m)$ equals $\text{True}$ if and only if $n \leq m$.

2.12. Let $f: A \to B$ be a function. We have to show that the relation $R \subseteq A^2$ given by $(x, y) \in R$ if and only if $f(x) = f(y)$ is an equivalence relation. This relation is reflexive, for $f(x) = f(x)$ certainly holds for all $x \in A$, and therefore $(x, x) \in R$ for all $x \in A$. Suppose $(x, y) \in R$. This means that $f(x) = f(y)$. Then also $f(y) = f(x)$, i.e. $(y, x) \in R$. So $R$ is symmetric. Finally, assume $(x, y) \in R$ and $(y, z) \in R$. Then $f(x) = f(y)$ and $f(y) = f(z)$. It follows that $f(x) = f(z)$, i.e. $(x, z) \in R$, and we have shown that $R$ is transitive.
2.13

\[(\lambda f \lambda x \mapsto f (f x)) \ (\lambda y \mapsto 1 + y)\]
\[\beta \rightarrow \lambda x \mapsto (\lambda y \mapsto 1 + y) \ ((\lambda y \mapsto 1 + y) \ x)\]
\[\beta \rightarrow \lambda x \mapsto (\lambda y \mapsto 1 + y) \ (1 + x)\]
\[\beta \rightarrow \lambda x \mapsto 1 + (1 + x)\]

2.14

\[(\lambda x \mapsto x \ x) \ (\lambda x \mapsto x \ x)\]
\[\beta \rightarrow (\lambda x \mapsto x \ x) \ (\lambda x \mapsto x \ x)\]
\[\beta \rightarrow \ldots\]

\[(\lambda x \mapsto x \ x \ x) \ (\lambda x \mapsto x \ x \ x)\]
\[\beta \rightarrow (\lambda x \mapsto x \ x \ x) \ (\lambda x \mapsto x \ x \ x) \ (\lambda x \mapsto x \ x \ x)\]
\[\beta \rightarrow \ldots\]

2.15

Dorothy likes Toto: t

Dorothy: e

(\lambda y \mapsto y \ likes \ Toto): e \rightarrow t

(\lambda x \lambda y \mapsto y \ likes \ x): e \rightarrow e \rightarrow t

Toto: e

2.16 It is not possible to find a finite type for \((\lambda x \mapsto x \ x) \ (\lambda x \mapsto x \ x)\). For it to have a type \(\tau\), it would have to hold that \((\lambda x \mapsto x \ x): \delta \rightarrow \tau\) and \((\lambda x \mapsto x \ x): \tau\) (according to the typing rule for application). So we could conclude that \(\tau\) equals \(\delta \rightarrow \tau\), which is not possible because the type system is not recursive. (You can make the same consideration thinking about the type of \(x\) in \(\lambda x \mapsto x \ x\).)

2.17 An appropriate type for very is \((N \rightarrow N) \rightarrow (N \rightarrow N)\). Then:
• \((\text{very}(N \rightarrow N) \rightarrow (N \rightarrow N)) \text{friendly}(N \rightarrow N) \) \(wizard_N\)
• \((\text{very}(N \rightarrow N) \rightarrow (N \rightarrow N) \rightarrow (N \rightarrow N)) \text{friendly}(N \rightarrow N) \) \(wizard_N\)

Note that the type \(N \rightarrow N\) also works for the example, with the bracketing:

\((\text{very friendly wizard})\).

But this makes the wrong prediction that \((\text{very}_{N \rightarrow N} wizard_N)\) is well-typed too.

---

**Solutions to Exercises from Chapter 3**

```haskell
module SolFPH where

import FPH

3.1 The operator \(^\) has precedence over \(*\) and \(/\), which take precedence over \(+\) and \(-\).

3.2 The interpreter reads it as \(2^{(3^4)}\), for we get:

```haskell
Prelude> 2^3^4
2417851639229258349412352
Prelude> 2^(3^4)
2417851639229258349412352
Prelude> (2^3)^4
4096
```

3.3 The general type of \((:):\) is \(a \rightarrow [a] \rightarrow [a]\). In this particular example it is used to put a character in front of a string, so the type is instantiated as

\[\text{Char} \rightarrow [\text{Char}] \rightarrow [\text{Char}]\].

3.4 \((>3)\) denotes the property of being greater than 3, i.e. \(\lambda x . x > 3\), and \((3>)\) denotes the property of being less than 3, i.e. \(\lambda x . 3 > x\).
3.5 `putStrLn (story (-1))` loops endlessly, because with every step the control variable \( k \) is decreased by 1 and thus the base case of 0 is never reached.

3.6 A recursive definition requires a base case, and the definition of GNU as ‘GNU’s not Unix’ does not have one. (This kind of recursive definition without a base case is called **co-recursion**.)

3.7 You might expect \( a \rightarrow a \rightarrow \text{Bool} \) as the type for \( \lambda x \, y \rightarrow x /= y \), but in fact you get \((\text{Eq} \ a) \Rightarrow a \rightarrow a \rightarrow \text{Bool}\). This is because this general type for inequality can only be instantiated by types whose instances can be checked for equality.

3.8 There is no difference. (The equivalence between expressions \( f \) and \( \lambda x \mapsto f x \) is called **\( \eta \)**-equivalence, and the reduction step from \( \lambda x \mapsto f x \) to \( f \) is called \( \eta \)-**conversion**.)

3.9 The type of the function composition \( \text{all} \ . \ (=/=) \) is

\[
(\text{Eq} \ a) \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}.
\]

The function checks whether an item of type \( a \) is unequal to all elements of a list of type \([a]\). An appropriate name would therefore be \texttt{notElem}, for **not element of**.

3.10 The type of \( \text{all} \ . \ (==) \) is the same as that of \( \text{all} \ . \ (=/=) \):

\[
(\text{Eq} \ a) \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}.
\]

An appropriate name for this function composition would be \texttt{elem}, for **element of**.

3.11 An algorithm for testing equality of infinite lists could proceed as follows. Test the first elements of both lists for equality. If they are not equal, return \texttt{False}, otherwise proceed with the rest of the list, until you find two elements that are not equal. Note that this way equality of infinite lists is only falsifiable but not verifiable in a finite number of steps. Here is an implementation (this works for both finite and infinite lists):
listEq :: (Eq a) => [a] -> [a] -> Bool
listEq [] []  = True
listEq [] _   = False
listEq _ []   = False
listEq (x:xs) (y:ys) = x == y && listEq xs ys

If the input lists are guaranteed to be infinite, the first three clauses can be dropped:

inflistEq :: (Eq a) => [a] -> [a] -> Bool
inflistEq (x:xs) (y:ys) = x == y && inflistEq xs ys

A call to inflistEq xs ys, where both xs and ys are infinite, will either yield False or run forever.

3.12

minList :: Ord a => [a] -> a
minList [x]   = x
minList (x:y:zs) = minList ((min x y) : zs)

Note that this function is predefined in the Haskell Prelude as minimum.

3.13

delete :: Ord a => a -> [a] -> [a]
delete x []    = []
delete x (y:ys) | x == y  = ys
            | otherwise = y : (delete x ys)

In order for it to delete every occurrence of x, the function has to be recursive also in the case of x == y, i.e. has to return delete x ys instead of ys.

3.14
srt :: Ord a => [a] -> [a]
srt [] = []
srt xs = x : srt (delete x xs)
    where x = minList xs

3.16

averageLength :: String -> Rational
averageLength sonnet = average (map f (words sonnet))
    where f = length . filter ('notElem' "'?;:,.")

3.17

sublist :: Ord a => [a] -> [a] -> Bool
sublist [] [] = True
sublist xs [] = False
sublist xs (y:ys) = prefix xs (y:ys) || sublist xs ys

3.18 The type of vh is Char -> Char.

3.19

data DeclClass = One | Two | Three | Four | Five

swedishPlural :: String -> DeclClass -> String
swedishPlural noun d = case d of
    One  -> init noun ++ "or"
    Two  -> init noun ++ "ar"
    Three -> if (last noun) 'elem' swedishVowels
             then noun ++ "r"
             else noun ++ "er"
    Four  -> noun ++ "n"
    Five  -> noun
appendSuffixY :: [Phoneme] -> [Phoneme] -> [Phoneme]
appendSuffixY stem suffix = stem ++ map (vh (vow stem)) suffix
   where
     vow     = head . filter ('elem' yawelmaniVowels)
     hi      = fValue High
     vh p p' | hi p == hi p' = (fMatch Back (fValue Back p)) . fMatch Round (fValue Round p)) p'
               | otherwise       = p'

Solutions to Exercises from Chapter 4

module SolFSynF where

import Data.List
import FSynF

4.1

column  →  A | B | C |  D |  E | F | G | H |  I | J
row     →  I |  2 | 3 |  4 |  5 |  6 |  7 |  8 |  9 | 10
attack  →  column row
ship     →  battleship | frigate | submarine | destroyer
reaction →  missed | hit ship | sunk ship
turn     →  attack reaction
surrender →  attack defeated
game     →  surrender | turn game
4.2

colour \longrightarrow red | yellow | blue | green | orange
answer \longrightarrow black | white
guess \longrightarrow colour colour colour colour
reaction \longrightarrow \{answer\}
turn \longrightarrow guess reaction
game \longrightarrow turn | turn turn | turn turn turn | turn turn turn turn

4.3 Grammar for chess:

figure \longrightarrow King | Queen | Knight | Rook | Bishop | Pawn
row \longrightarrow \textit{a} | \textit{b} | \textit{c} | \textit{d} | \textit{e} | \textit{f} | \textit{g} | \textit{h}
column \longrightarrow \textit{1} | \textit{2} | \textit{3} | \textit{4} | \textit{5} | \textit{6} | \textit{7} | \textit{8}
whitemove \longrightarrow figure row column
blackmove \longrightarrow figure row column
game \longrightarrow whitemove blackmove game

The grammar for bingo is left to the reader.

4.4 The sure sign of an infinite context-free language is a production of the form $A \rightarrow WAV$, where $W$ and $V$ are not both empty. An example is the production rule $\text{game} \rightarrow \text{turn game}$. This is called recursive use of a nonterminal. The recursion can also be indirect, via one or more other nonterminals: $A \rightarrow WBV$, $B \rightarrow YAZ$ is an example of recursive use of $A, B$. Check the grammars for this kind of recursion and you have spotted the ones that generate infinite languages.

4.5 Introduce a rewrite symbol $\text{eps}$ for the empty string:

character \longrightarrow A | \cdots | Z | a | \cdots | z | _ | , | . | ? | ! | ; | :
string \longrightarrow \text{eps} | character string
eps \longrightarrow
4.6 We add a rule for adjectives (ADJ) and extend the rule for CNs with the possibility of having an adjective in front of a common noun (note that this is recursive and allows for NPs like *the happy evil dwarf*).

\[
\begin{align*}
\text{ADJ} & \rightarrow \text{happy} \mid \text{evil} \\
\text{CN} & \rightarrow \ldots \mid \text{ADJ} \text{CN}
\end{align*}
\]

4.7 First we add two rules for building PPs from a preposition and an NP. Then we extend the rule for NPs, in order to generate NPs like *a giant with a sword*. Note that we don’t simply add a recursive production \( \text{NP} \rightarrow \text{NP PP} \). If we did this, we would not only allow arbitrary many PPs as NP modifiers but also generate NPs like *Little Mook with a sword* (which we want to exclude). Finally, we also extend the VP rule for building VPs with preposition phrases.

\[
\begin{align*}
\text{P} & \rightarrow \text{with} \\
\text{PP} & \rightarrow \text{P} \text{ NP} \\
\text{NP} & \rightarrow \ldots \mid \text{DET} \text{ CN} \text{ PP} \\
\text{VP} & \rightarrow \ldots \mid \text{TV} \text{ NP} \text{ PP}
\end{align*}
\]

4.8

We extend the fragment with the following productions:

\[
\begin{align*}
\text{COORD} & \rightarrow \text{and} \\
\text{RCN} & \rightarrow \ldots \mid \text{CN that} \text{ VP} \text{ COORD} \text{ VP} \\
& \mid \text{CN that} \text{ NP} \text{ TV} \text{ COORD} \text{ NP} \text{ TV}
\end{align*}
\]

These rules take into account that only ‘parallel’ RCNs may be coordinated (cf. [Ros67]): while *the dwarf that helped Goldilocks and admired Snow White* is fine, an NP like *the dwarf that Goldilocks helped and admired Snow White* is ungrammatical.

However, they have the disadvantage that they are not recursive, so we cannot generate *the dwarf that helped Goldilocks and admired the princess that shuddered*...
and laughed, unless we add more productions. We can do this, of course, but we seem to miss a generalisation.

4.12 Assume $p$ expresses the wizard polishes his wand, $q$ expresses the wizard learns a new spell, $r$ expresses the wizard is lazy, $s$ expresses the peasant will deal with the devil, $t$ expresses the peasant has a plan to outwit the devil, $u$ expresses unicorns exist, $v$ expresses dragons exist, and $w$ expresses goblins exist. Then we can translate the sentences as follows.

The wizard polishes his wand and learns a new spell, or he is lazy. (4.1)
\[ \sim (p \land q) \lor r \]

The peasant will deal with the devil only if he has a plan to outwit him. (4.2)
\[ \sim (s \land \neg t) \]

If neither unicorns nor dragons exist, then neither do goblins. (4.3)
\[ \sim (\neg u \land \neg v \land w) \]

The problem was to find reasonable translations for only if and for if . . . then.

4.13 Define $F_1 \oplus F_2$ as $(F_1 \land \neg F_2) \lor (\neg F_1 \land F_2)$.

4.14 Formulas of propositional logic in Polish notation are uniquely readable. The atoms are surely uniquely readable. Assume that $P$ is uniquely readable. Then $\sim P$ is as well. Assume that $P_1$ and $P_2$ are uniquely readable. Then $\land P_1 P_2$ and $\lor P_1 P_2$ are uniquely readable as well.

4.15

\begin{verbatim}
opsNr :: Form -> Int
opsNr (P _) = 0
opsNr (Ng f) = 1 + opsNr f
opsNr (Cnj fs) = 1 + sum (map opsNr fs)
opsNr (Dsj fs) = 1 + sum (map opsNr fs)
\end{verbatim}

4.16
depth :: Form -> Int
depth (P _) = 0
depth (Ng f) = 1 + depth f
depth (Cnj []) = 1
depth (Cnj fs) = 1 + maximum (map depth fs)
depth (Dsj []) = 1
depth (Dsj fs) = 1 + maximum (map depth fs)

4.17

propNames :: Form -> [String]
propNames (P name) = [name]
propNames (Ng f) = propNames f
propNames (Cnj fs) = (sort.nub.concat) (map propNames fs)
propNames (Dsj fs) = (sort.nub.concat) (map propNames fs)

4.18

Solutions to Exercises from Chapter 5

module SolFSemF where

import FSynF
import FSemF
import InfEngine
import System.Random

Note: System.Random is used for random number generation in the Mastermind implementation.

5.1

As the picture on page ?? suggests, a battleship takes up five squares, a frigate four,
a destroyer three, and a submarine two. To check whether a vessel is sunk by an attack, find out whether find out if the ship occu

sunk :: Attack -> Ship -> State -> Bool
sunk (row,column) (positions,earlierHits) Battleship
sunk (row,column) Frigate
sunk (row,column) Submarine
sunk (row,column) Destroyer

5.2
There are lots of other things to be said. E.g., you should not shout ‘hit’ as a response to an attack when this is not true (Quality). You should not keep quiet as a response to an attack (Quantity). You should not mumble something incomprehensible as a response to an attack (Mode of Expression). When voicing an attack, you should speak up loud and clear (Mode of Expression). In short, the Gricean maxims tell you to play the game according to the rules, and they also tell you a thing or two about how to interpret the rules.

5.3

(1) \( V^+ (\neg p \lor p) = 1 \),
(2) \( V^+ (p \land \neg p) = 0 \),
(3) \( V^+ (\neg (p \land \neg r)) = 0 \),
(4) \( V^+ ((p \land \neg r)) = 1 \),
(5) \( V^+ (p \lor (q \land r)) = 1 \).

5.4 A tautology is a formula that is true for any valuation. If you negate a tautology you get a formula that is false for any valuation. By definition, this is a contradiction. Vice versa the same holds, so any negated contradiction is a tautology.

5.5

(1) \( p \land \neg q \) is satisfied by \( p \mapsto 1, q \mapsto 0 \).
(2) \( p \land \neg p \) is not satisfiable.
(3) \( p \rightarrow \neg p \) is satisfied by \( p \mapsto 0 \).

5.6
(1) \( \neg
\neg p \equiv p \) is true.

(2) \( p \rightarrow q \equiv \neg p \vee q \) is true.

(3) \( \neg(p \leftrightarrow q) \equiv \neg p \land q \) is false.

5.7

(1) \( p \models p \lor q \) is true.

(2) \( p \rightarrow q \models \neg p \rightarrow \neg q \) is false.

(3) \( \neg q \models p \rightarrow q \) is false.

(4) \( \neg p, q \rightarrow p \models \neg q \) is true.

5.8 Assume \( F_1 \models F_2 \). This means that every valuation that makes \( F_1 \) true makes \( F_2 \) true. This in turn means that every valuation that makes \( F_2 \) false makes \( F_1 \) false. But this means that every valuation that makes \( \neg F_2 \) true makes \( \neg F_2 \) true. In other words, \( \neg F_2 \models \neg F_1 \).

5.10

\[
\text{impliesL :: [Form] -> Form -> Bool} \\
\text{impliesL = implies . Cnj}
\]

5.11

\[
\text{propEquiv :: Form -> Form -> Bool} \\
\text{propEquiv f1 f2 = implies f1 f2 && implies f2 f1}
\]

5.12
altEval :: [String] -> Form -> Bool
altEval [] (P c) = False
altEval (i:xs) (P c)
    | c == i = True
    | otherwise = altEval xs (P c)
altEval xs (Ng f) = not (altEval xs f)
altEval xs (Cnj fs) = all (altEval xs) fs
altEval xs (Ds) = any (altEval xs) fs

5.13 If colour repetition is allowed there are $5^4$ possible settings: five colour choices for the first position times five colour choices for the second position times five colour choices for the third position times five colour choices for the fourth position. If colour repetition is forbidden there are $5 \times 4 \times 3 \times 2$ settings left. Five choices for the first position, one choice less for the second position because the colour of the first position is ruled out for the second position, and so on.

5.14 The formula for the first position:

$r_1 \leftrightarrow \neg(y_1 \lor b_1 \lor g_1 \lor o_1)$
\[\land\]
$y_1 \leftrightarrow \neg(r_1 \lor b_1 \lor g_1 \lor o_1)$
\[\land\]
$b_1 \leftrightarrow \neg(r_1 \lor y_1 \lor g_1 \lor o_1)$
\[\land\]
$g_1 \leftrightarrow \neg(r_1 \lor y_1 \lor b_1 \lor o_1)$
\[\land\]
$o_1 \leftrightarrow \neg(r_1 \lor y_1 \lor b_1 \lor g_1)$.

The formula we are after is the conjunction of the formulas for the four positions.

5.15

First include the function that was given:

getc = 
getColours :: IO [Colour]
getc = do
    i <- getStdRandom (randomR (0,4))
    j <- getStdRandom (randomR (0,4))
    k <- getStdRandom (randomR (0,4))
    l <- getStdRandom (randomR (0,4))
    return [toEnum i, toEnum j, toEnum k, toEnum l]
Next, define a game for a given secret:

```haskell
playgame :: [Colour] -> IO()
playgame secret =
  do
    putStrLn "Give a sequence of four colours from RGBYO"
    str <- getLine
    let guess = string2pattern str
    in if guess /= secret
        then let answer = reaction secret guess
            in do
                  putStrLn (show answer)
                  putStrLn "Please make another guess"
                  playgame secret
            else putStrLn "correct"

Finally, the function that generates a secret and plays the game for that secret.

```haskell
mm :: IO ()
mm = do
  secret <- getColours
  playgame secret
```

5.16

Stupid guesses are the guesses that are already ruled out by previous feedback. To check this, we have to keep track of the information state. The `stupid` function checks the guess against the current state, as follows.

```haskell
stupid :: [Pattern] -> Pattern -> Bool
stupid state guess = notElem guess state
```

The initial information state is given by:
startState :: [Pattern]
startState = let colours = [minBound..maxBound] in
             [ [c1,c2,c3,c4] | c1 <- colours, c2 <- colours,
                          c3 <- colours, c4 <- colours ]

New version of the Mastermind game that uses this:

play :: IO()
play = play0 startState

play0 :: [Pattern] -> IO()
play0 state =
do
  putStrLn "Give a sequence of four colours from RGBYO"
  str <- getLine
  let guess = string2pattern str
  in do
    if stupid state guess
     then putStrLn "Not very clever"
     else putStrLn "Hmm, clever guess.."
    if guess /= secret
     then let answer = reaction secret guess
             in do
              putStrLn (show answer)
              putStrLn "Please make another guess"
              play0 (updateMM state guess answer)
     else putStrLn "correct"

5.17

(1) To see that $\forall x (A x \land B x)$ means something stronger than All A are B, consider a model with $A = \{a\}, B = \{a, b\}$. In this model, All A are B is true, but $\forall x (A x \land B x)$ is false, for $b$ is a counterexample: an element of the domain of discourse that is $B$ but not $A$.

(2) To see that $\exists x (A x \rightarrow B x)$ means something weaker than Some A are B,
consider a model with $A = \emptyset$ and $B = \{b\}$. Then $b$ has the property $\lambda x. Ax \rightarrow B x$, so $\exists x (Ax \rightarrow Bx)$ is true. But Some $A$ are $B$ is false in this model.

5.18

*Someone walks and someone talks.*  
\[
\exists x (\text{Person } x \land \text{Walk } x \land \text{Talk } x)
\]

*No wizard cast a spell or mixed a potion.*  
\[
\neg \exists x (\text{Wizard } x \land \exists y \exists z ((\text{Spell } y \land \text{Cast } x y) \lor (\text{Potion } z \land \text{Mix } x z)))
\]

*Every balad that is sung by a princess is beautiful.*  
\[
\forall x \forall y ((\text{Ballad } x \land \text{Princess } y \land \text{Sing } y x) \rightarrow \text{Beautiful } x)
\]

*If a knight finds a dragon, he fights it.*  
\[
\forall x \forall y ((\text{Knight } x \land \text{Dragon } y \land \text{Find } xy) \rightarrow \text{Fight } x y)
\]

5.19

(1) $M \models \exists x (P x \land R x x)$ is true.

(2) $M \models \forall x (P x \rightarrow \exists y R x y)$ is true.

(3) $M \models \forall x (\exists y R y x \rightarrow R x x)$ is false.

5.20

(1) $\models \forall x P x \lor \exists x \neg P x$ is true.

(2) $\models \exists x \exists y R x y \rightarrow \exists x \exists y R y x$ is false.

(3) $\models \forall x R x x \rightarrow \forall x \exists y R x y$ is true.

(4) $\models \exists x R x x \rightarrow \forall x \exists y R x y$ is false.
5.21 Define $F^v_a$ as follows.

\[
(P(t))^v_a := P(t^v_a)
\]

\[
(R(t_1, t_2))^v_a := R(t_1^v_a, t_2^v_a)
\]

\[
(S(t_1, t_2, t_3))^v_a := S(t_1^v_a, t_2^v_a, t_3^v_a)
\]

\[
(\neg F)^v_a := \neg F^v_a
\]

\[
(F_1 \wedge F_2)^v_a := F_1^v_a \wedge F_2^v_a
\]

\[
(F_1 \vee F_2)^v_a := F_1^v_a \vee F_2^v_a
\]

\[
(\forall v F)^v_a := \forall v F
\]

\[
(\forall u F)^v_a := \forall u F^v_a \text{ for } u \text{ different from } v
\]

\[
(\exists v F)^v_a := \exists v F
\]

\[
(\exists u F)^v_a := \exists u F^v_a \text{ for } u \text{ different from } v
\]

5.22 Assume that model $M$ has a domain $D$ with a name $\hat{d}$ for each $d \in D$. Use $F^v_d$ in the new version of the truth definition, where truth is defined for all closed formulas of the language. Note that no variable assignments are needed. Note also that $a, a_1, \ldots$ are used for constants of the language; this includes the new names of the form $\hat{d}$:

\[
M \models P(a) \iff I(a) \in I(P)
\]

\[
M \models R(a_1, a_2) \iff (I(a_1), I(a_2)) \in I(R)
\]

\[
M \models S(a_1, a_2, a_3) \iff (I(a_1), I(a_2), I(a_3)) \in I(S)
\]

\[
M \models a_1 = a_2 \iff I(a_1) = I(a_2)
\]

\[
M \models \neg F \iff \text{it is not the case that } M \models F
\]

\[
M \models (F_1 \wedge F_2) \iff M \models F_1 \text{ and } M \models F_2
\]

\[
M \models (F_1 \vee F_2) \iff M \models F_1 \text{ or } M \models F_2
\]

\[
M \models \forall v F \iff \text{for all } d \in D \text{ it holds that } M \models F^v_d
\]

\[
M \models \exists v F \iff \text{for at least one } d \in D \text{ it holds that } M \models F^v_d
\]

5.23

5.24

(1) $\forall x P x \models \exists x P x$ does not hold. On the model with an empty domain, $\forall x P x$
is true, but $\exists x P x$ is false. If we demand that all models have a non-empty domain of discourse, then the statement holds.

(2) $\exists x \exists y R x y \models \exists x R x x$ does not hold. Consider a model with domain $\{a, b\}$, where $R$ is interpreted as $\{(a, b)\}$. Then $\exists x \exists y R x y$ is true, but $\exists x R x x$ is false.

(3) $\exists y \forall x R x y \models \forall x \exists y R x y$ does hold. If there is something that every woman desires, then for every woman there is something she desires.

5.25

(1) $\forall x \forall y (R x y \rightarrow R y x), \ R a b \models R b a$
   This holds. The first premise states that $R$ is symmetric. So if $R a b$ is true, $R b a$ has to be true as well.

(2) $\forall x \forall y (R x y \rightarrow R y x), \ R a b \models R a a$
   This does not hold. The first premise states that $R$ is symmetric. But symmetry does not imply reflexivity. Consider a model with domain $\{a, b\}$ and with $R$ interpreted as $\{(a, b), (b, a)\}$. Then both premises are true in this model, but the conclusion is false.

5.26

We have seen already that universal statements are translated using implications. The formula says that it holds for everything in the domain of discourse that if it is a boy then there exists a girl that he loves. This correctly expresses the meaning of *Every boy loved a girl*, if we disregard tense.

5.27

Every girl that laughed helped a boy. \hspace{1cm} (5.8)

$\forall x ((\text{Girl } x \land \text{Laugh } x) \rightarrow \exists y (\text{Boy } y \land \text{Help } x y))$

No giant that shuddered killed every dwarf. \hspace{1cm} (5.9)

$\neg \exists x ((\text{Giant } x \land \text{Shudder } x) \land \forall y (\text{Dwarf } y \rightarrow \text{Kill } x y))$

Every princess loved every dwarf that killed a giant. \hspace{1cm} (5.10)

$\forall x \forall y ((\text{Princess } x \land \text{Dwarf } y \land \exists z (\text{Giant } z \land \text{Kill } y z)) \rightarrow \text{Love } x y)$

Every boy admired a girl that no wizard helped. \hspace{1cm} (5.11)

$\forall x (\text{Boy } x \rightarrow \exists y (\text{Girl } y \land \neg \exists z (\text{Wizard } z \land \text{Help } z y) \land \text{Admire } x y))$
Note: there are other solutions, but these are logically equivalent to the formulas given here. This is because the same content can be expressed in predicate logic with different formulas.

5.28

5.29

5.30

5.31

5.32 Mention of a class in a fact:

mention :: Class -> (Class, Class, Bool) -> Bool
mention xs (ys, zs, _) =
  elem xs [ys,zs] || elem (opp xs) [ys,zs]

Filter the facts from the knowledge base that mention a class $A$:

filterKB :: Class -> KB -> KB
filterKB xs = filter (mention xs)

Report on a class $A$ by listing what the knowledge base says about $A$:

report :: KB -> Class -> [Statement]
report kb as = map f2s (filterKB as kb)
module SolMCWPL where

import Model
import MCWPL

6.2

The expression \((\text{passivize} \text{ admire}) \ G\) checks whether the pair \((\text{Unspec},G)\) is in the denotation of \text{admire}, and it is not. However, we want to infer that \((\text{Unspec},G)\) is in the relation if \((x,G)\) with some entity \(x\) is in the relation. Up to now, our implementation does not know how to do this inference, so we need to tell it. One way to do that is to build the closure of a relation under moving from some entity \(x\) to \text{Unspec}:

\[
\text{unspecClose} :: [(\text{Entity},\text{Entity})] \rightarrow [(\text{Entity},\text{Entity})]
\]

\[
\text{unspecClose} \ r = r ++ \left[ (\text{Unspec},y) \mid x \leftarrow \text{entities}, \ y \leftarrow \text{entities}, \ (x,y) \ 'elem' \ r \right]
\]

\[
++ \left[ (x,\text{Unspec}) \mid x \leftarrow \text{entities}, \ y \leftarrow \text{entities}, \ (x,y) \ 'elem' \ r \right]
\]

Then we can define a relation like \text{admire} as follows:

\[
\text{admire} = \text{curry} \ (('elem' \ (\text{unspecClose} [(x,G) \mid x \leftarrow \text{entities}, \ \text{person} \ x])))
\]

6.3

\[
\text{passivize} :: \text{ThreePlacePred} \rightarrow \text{TwoPlacePred}
\]

\[
\text{passivize} \ r = \ \backslash \ x \ y \rightarrow r \ \text{Unspec} \ x \ y
\]
checkSentence :: S -> Bool
checkSentence s = evl entities intPreds ...

intPreds :: String -> [Entity] -> Bool
intPreds "girl" = girl

---

Solutions to Exercises from Chapter 7

module SolTCOM where

import TCOM

---

7.1

tree :: Integer -> [(Integer,Integer)]
tree n = [(n-x,x) | x <- [0..n]]

treeOfNumbers :: [(Integer,Integer)]
treeOfNumbers = concat [ tree n | n <- [0..] ]

---

7.2

7.3
type Quant = (Integer -> Bool) -> [Integer] -> Bool

check :: Quant -> (Integer,Integer) -> Bool
check q (n,m) = q (\ x -> 0 < x && x <= m) [1..n+m]

genTree :: Quant -> [(Integer,Integer)]
genTree q = filter (check q) treeOfNumbers

Here are two example applications, using the predefined quantifiers from Haskell:

SolTCOM> take 10 (genTree all)
[(0,0),(0,1),(0,2),(0,3),(0,4),(0,5),(0,6),(0,7),(0,8),(0,9)]
SolTCOM> take 10 (genTree any)
[(0,1),(1,1),(0,2),(2,1),(1,2),(0,3),(3,1),(2,2),(1,3),(0,4)]
8.1 A cow would still have four legs. People in the hypothetical situation would say “five”, but that does not matter to us. We know how to call a tail.
28
8.7

\textbf{cnINT} :: \text{CN} \to \text{World} \to \text{Entity} \to \text{Bool}
\text{cnINT} \text{ Girl} = \text{iGirl}
\text{cnINT} \text{ Princess} = \text{iPrincess}

\textbf{intensCN} :: \text{CN} \to \text{IEntity} \to \text{IBool}
\text{intensCN} = \text{iProp} \cdot \text{cnINT}

8.8

\textbf{npINT} :: \text{NP} \to \text{World} \to \text{(Entity} \to \text{Bool}) \to \text{Bool}
\text{npINT} \text{ np} = \ \lambda \text{i} \to \text{intNP} \text{ np}

\textbf{intensNP} :: \text{NP} \to \text{(IEntity} \to \text{IBool}) \to \text{IBool}
\text{intensNP} = \text{iPropToB} \cdot \text{npINT}

\textbf{Solutions to Exercises from Chapter 9}

\textbf{module} \text{SolP where}

\textbf{import} \text{ P}

9.1

\textbf{immdominance} :: \text{ParseTree} \ a \ b \to \text{Rel} \ \text{Pos}
\text{immdominance} \text{ t} = [ (p,q) \mid (p,q) \leftarrow \text{properdominance} \text{ t},
\text{not} \ (\text{any} \ (\text{inbetween} \ p \ q) \ \text{(pos} \ \text{t})) ]
\text{where}\ \text{inbetween} \ p \ q \ r = (p,r) \text{‘elem’} \ \text{(properdominance} \text{ t})
\text{&} (r,q) \text{‘elem’} \ \text{(properdominance} \text{ t})
9.2

For (1), assume that \( p \) dominates \( q \). Then clearly, no sister of \( p \) dominates \( q \). It follows that \( p \) does not c-command \( q \). Thus, (1) follows from \( p \) c-commands \( q \) by contraposition.

For (2), we reason again by contraposition. Suppose \( q \) dominates \( p \). Then there is no sister of \( p \) that dominates \( q \). Therefore, \( p \) does not c-command \( q \).

Same recipe for (3). Assume the lowest branching node that dominates \( p \) does not dominate \( q \). Then no sister of \( p \) dominates \( q \). Therefore, \( p \) does not c-command \( q \).

9.3

\[
\text{mutualCCommand} :: \text{ParseTree} \ a \ b \rightarrow \text{Rel Pos}\\
\text{mutualCCommand} \ t = [ (p,q) \mid (p,q) \leftarrow \text{cCommand} \ t,\ (q,p) \ 'elem' \ \text{cCommand} \ t ]
\]

Here is why mutual c-command and sisterhood coincide:

Assume \( p \) and \( q \) c-command each other. Since \( p \) c-commands \( q \), \( p \) has a sister \( r \) that dominates \( q \). Moreover, it has to hold that \( r \) and \( q \) are the same: if \( r \) would properly dominate \( q \), then \( q \) could not have a sister that dominates \( p \), which contradicts the fact that \( q \) does have such a sister because it c-commands \( p \). Therefore \( r = q \), and since \( p \) and \( r \) are sisters, \( p \) and \( q \) are sisters.

The other direction is even more straightforward: if \( p \) and \( q \) are sisters, then there trivially exists a position that is sister of \( p \) and dominates \( q \) (namely \( q \) itself), and a position that is sister of \( q \) and dominates \( p \) (namely \( p \) itself). Thus, \( p \) and \( q \) c-command each other.

9.4

A position \( p \) immediately precedes a position \( q \) in a tree if \( p \) precedes \( q \) and there is no position \( r \) such that \( p \) precedes \( r \) and \( r \) precedes \( q \).
immprecedence :: ParseTree a b -> Rel Pos
immprecedence t = (p,q) <- precedence t,
    not (any (inbetween p q) (pos t))
where inbetween p q r = (p,r) 'elem' (precedence t)
    && (r,q) 'elem' (precedence t)

command :: ParseTree a b -> Rel Pos
command t = [ (p,q) | p <- pos t, q <- pos t,
    (p,q) 'notElem' (dominance t),
    (q,p) 'notElem' (dominance t),
    (p,[]) 'elem' (dominance t),
    ([q],p) 'elem' (dominance t),
    ([q],q) 'elem' (dominance t) ]
Solutions to Exercises from Chapter 10

module SolHRAS where

import HRAS

10.1
10.2
10.3
10.4
10.5

Solutions to Exercises from Chapter 11

module SolCPSS where

import CPSS

11.1

Both are of type \(((e \to t) \to t) \to t\).

\[
\begin{align*}
\llbracket \text{helped Dorothy} \rrbracket &= \lambda Q \mapsto (Q (\text{Help } d)) \\
\llbracket \text{Alice helped} \rrbracket &= \lambda Q \mapsto (Q (\lambda x \mapsto ((\text{Help } x) a)))
\end{align*}
\]
11.2

The types for adjectives are the following:

<table>
<thead>
<tr>
<th>value type</th>
<th>continuation type</th>
<th>computation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>((e \rightarrow t))</td>
<td>((((e \rightarrow t) \rightarrow (e \rightarrow t)) \rightarrow t))</td>
<td>(((((e \rightarrow t) \rightarrow (e \rightarrow t)) \rightarrow t) \rightarrow t))</td>
</tr>
</tbody>
</table>

11.3

In order to compute the meaning of *Everyone admired Goldilocks*, first compute the meaning of the VP *admired Goldilocks*. This is parallel to the meaning computation of *helped Dorothy* on page 299, so we just give the result:

\[
\begin{align*}
\cdot \text{[admired Goldilocks]} & = \lambda k \mapsto (k \text{ (Admire } g )) \\
\end{align*}
\]

Next, this VP meaning is applied to the meaning of *everyone*:

\[
\begin{align*}
\lambda k \mapsto \left( \lambda n \mapsto \left( \lambda m \mapsto (k \text{ (Admire } n )) \right) \right)
= \lambda k \mapsto \left( \lambda n \mapsto \left( \lambda m \mapsto (k \text{ (Admire } n )) \right) \right)
\end{align*}
\]

In order to compute the meaning of *Goldilocks admired someone*, again start with computing the VP meaning:

\[
\begin{align*}
\cdot \text{[admired]} & = \lambda k \mapsto \left( \lambda n \mapsto \left( \lambda m \mapsto (k \text{ (Admire } n )) \right) \right)
= \lambda k \mapsto \left( \lambda n \mapsto \left( \lambda m \mapsto (k \text{ (Admire } n )) \right) \right)
\end{align*}
\]

Next, apply this VP meaning to the meaning of *Goldilocks* (we use \((P \; x)\) as ab-
abbreviation of \((\text{Person } x)\) and \((A \ x)\) as abbreviation of \((\text{Admire } x)\):

\[
\begin{align*}
\text{admired someone} & \equiv \lambda x \rightarrow (\text{Goldilocks} \ (\lambda n \rightarrow (\text{admired someone} \ (\lambda m \rightarrow (k \ (m \ n)))))) \\
\text{Goldilocks} & = \lambda x \rightarrow (\text{Admire } x) \ ((\lambda x \rightarrow (J \ x) \ (\lambda y \rightarrow (A \ y)) \ (\lambda n \rightarrow (k \ (m \ n))))))) \\
\beta & \rightarrow \lambda x \rightarrow (\text{Goldilocks} \ (\lambda n \rightarrow \exists y ((P \ x) \ (k \ (A \ y))))) \\
& = \lambda x \rightarrow ((\lambda k' \rightarrow (k' \ y)) \ (\lambda n \rightarrow \exists y ((P \ x) \ (k \ (A \ y))))) \\
\beta & \rightarrow \lambda x \rightarrow \exists y ((P \ x) \ (k \ (A \ y)))
\end{align*}
\]

11.6

Again we use \((P \ x)\) as abbreviation of \((\text{Person } x)\).

\[
\begin{align*}
\text{helped someone} & \equiv \lambda x \rightarrow (\text{someone} \ (\lambda m \rightarrow ((\lambda x \rightarrow (J \ x) \ (\lambda n \rightarrow (k \ (m \ n))))))) \\
\text{someone} & = \lambda x \rightarrow ((\lambda k' \rightarrow (k' \ y)) \ (\lambda n \rightarrow (k \ (m \ n)))))) \\
\beta & \rightarrow \lambda x \rightarrow ((\lambda k' \rightarrow (k' \ y)) \ (\lambda n \rightarrow (k \ (m \ n)))) \\
& = \lambda x \rightarrow ((\lambda k' \rightarrow (k' \ y)) \ (\lambda n \rightarrow (k \ (m \ n)))) \\
\beta & \rightarrow \lambda x \rightarrow \exists y ((P \ x) \ (k \ (A \ y)))
\end{align*}
\]

\[
\begin{align*}
\text{helped everyone} & \equiv \lambda x \rightarrow (\text{everyone} \ (\lambda m \rightarrow ((\lambda x \rightarrow (J \ x) \ (\lambda n \rightarrow (k \ (m \ n))))))) \\
\text{everyone} & = \lambda x \rightarrow ((\lambda k' \rightarrow (k' \ y)) \ (\lambda n \rightarrow (k \ (m \ n)))))) \\
\beta & \rightarrow \lambda x \rightarrow ((\lambda k' \rightarrow (k' \ y)) \ (\lambda n \rightarrow (k \ (m \ n)))) \\
& = \lambda x \rightarrow ((\lambda k' \rightarrow (k' \ y)) \ (\lambda n \rightarrow (k \ (m \ n)))) \\
\beta & \rightarrow \lambda x \rightarrow \exists y ((P \ y) \ (\lambda n \rightarrow (k \ (A \ y))))
\end{align*}
\]
Solutions to Exercises from Chapter 12

module SolDRAC where
import DRAC

12.1
12.2
12.3
12.4
12.5
12.6
12.7

Solutions to Exercises from Chapter 13

module SolCAIA where
import List
import CAIA

13.1 No, it does not. The machine counts the money before you are able to count it yourself, so the machine does not know whether you know that the amount is correct.

13.2 The fourth man learns that his cap must have the same colour as that of the third man, and that the first two guys wear caps of the other colour.
13.3

13.4

13.5

13.6

13.7

13.8 The union of two equivalence relations need not itself be an equivalence relation. Consider the following case:

\( R_a = \{(1,1), (2,2), (3,3), (1,2), (2,1)\}, \quad R_b = \{(1,1), (2,2), (3,3), (2,3), (3,2)\} \)

Then these two sets are equivalences, but their union is not an equivalence, for \( R_a \cup R_b \) is not transitive:

\[ R_a \cup R_b = \{(1,1), (2,2), (3,3), (1,2), (2,1), (2,3), (3,2)\} \]

13.9

The four wise men are \( a, b, c, d \). Let \( q_i \) express that the man in \( i \)-th position is wearing a white cap.

\[
\begin{align*}
q_1, q_2, q_3, q_4 &:: \text{Form} \\
q_1 &= \text{Prop (Q 1)}; \quad q_2 = \text{Prop (Q 2)} \\
q_3 &= \text{Prop (Q 3)}; \quad q_4 = \text{Prop (Q 4)}
\end{align*}
\]

Let’s picture the initial situation where the caps of the first man and the third man are white. To capture the information about what each man can see we can use the \texttt{computeAcc} function. Note that the propositions listed are the propositions that each agent has \textit{no} information about:
initWise :: EpistM Integer
initWise =
  Mo states
  [a..d]
  valuation
  (computeAcc a states [Q 1, Q 2, Q 3, Q 4] valuation
   ++
   computeAcc b states [Q 2, Q 3, Q 4] valuation
   ++
   computeAcc c states [Q 3, Q 4] valuation
   ++
   computeAcc d states [Q 1, Q 2, Q 3, Q 4] valuation)
[10]
where
states = [0..15]
valuation = zip states (powerList [Q 1, Q 2, Q 3, Q 4])

The following caps info formula expresses that exactly two of the four caps are white.

capsInfo :: Form
capsInfo =
  Disj [Conj [f, g, Neg h, Neg j] |
    f <- [q1, q2, q3, q4],
    g <- [q1, q2, q3, q4] \ [f],
    h <- [q1, q2, q3, q4] \ [f,g],
    j <- [q1, q2, q3, q4] \ [f,g,h],
    f < g, h < j]

This gives:

v & [q1,q2,-q3,q4],&[q1,q3,-q2,-q4],
&[q1,q4,-q2,-q3],&[q2,q3,-q1,-q4],
&[q2,q4,-q1,-q3],&[q3,q4,-q1,-q2]

Update of initial model with caps info:
mo1 = convert (upd_pa initWise capsInfo)

The statement that the third man (c) does know his cap colour:

cKnows = Disj [K (Agent c) q3, K (Agent c) (Neg q3)]

Update with the information that the third man (c) does not know his cap colour:

mo2 = convert (upd_pa mo1 (Neg cKnows))

Now we have to check whether b knows the colour of his cap:

bKnows = Disj [K (Agent b) q2, K (Agent b) (Neg q2)]

test = isTrue mo2 bKnows

Here is the answer:

SolCAIA> test
True

13.10

First some conventions for the representation of the basic facts. Use r1 for m, r2 for a, r3 for u.

male = Prop (R 1)
adult = Prop (R 2)
unmarried = Prop (R 3)

Let the two agents be alice and bob. Then the initial situation is like this:
38
13.11
13.12
13.13