School of Computing and Information Systems
COMP30026 Models of Computation

Assignment 1

Released: 26 Aug. 2022; Deadline: 16 Sep. 2022

Aims & Procedure

One aim of this assignment is to improve and test your understanding of propositional logic and first-order predicate logic, including their use in mechanised reasoning. A second aim is to develop your skills in analysis and formal reasoning about complex concepts, and to practise writing down formal arguments with clarity.

This document is the assignment spec. There are six challenges which you will find in the remainder of this document. Your answers to Challenge 5 and Challenge 6 are to be submitted through Gradescope, for which there is a menu item on Canvas. The remaining challenges are to be completed on Grok, where you will find a module called "Assignment 1". That module also contains more detailed information about submission formats and how to submit your answers in Grok.

You are required to solve the challenges *individually*. You will probably find them to be of varying difficulty, but each is worth 2 marks, for a total of 12.

Challenge 1 – Predictably Inconsistent Weather

The city of Melbourne, Australia is infamous for its predictably inconsistent weather. The mobile apps Parrot and Carrot compete to predict the correct weather over the next three days. Melbourne's weather has a habit of making a fool of the apps' developers, such that at any time, only one of the two apps makes a correct prediction.

Despite this, Harald still can use this information to get accurate weather forecasts for the week, so that they don't get wet on their commutes to-and-from university. On a Monday, Harald checks both the Carrot and Parrot apps, where they make the following predictions:

Carrot:	"It will rain on Tuesday and Wednesday."
Parrot:	"If it rains on Monday, it will rain on Wednesday."

Task 1A. Capture, as a single propositional formula, the information that was thereby available to Harald. You will need to take into account which app makes each prediction. Use propositional letters as follows:

C :	Carrot's prediction is correct	P :	Parrot's prediction is correct		
M :	It rains on Monday	T :	It rains on Tuesday	W :	It rains on Wednesday

Task 1B. Harald tries to determine the weather forecast for the week from those two predictions, but realises they do not yet have enough information. Determine which truth assignments to C, P, M, T, W make your formula from Task 1A true.

Task 1C. Harald opens their window blinds and looks outside to check for any chance of rain. Based on *that* information, they now knew exactly what the weather would be for Monday, Tuesday and Wednesday. Given this information, determine, for each of Monday, Tuesday and Wednesday, whether it rains or not.

Submission and Marking: Your answer should be submitted on Grok. You will find the submission format explained there. You will receive some feedback from some elementary tests. These merely check that your input has the correct format; they should not be relied upon for correctness. We will test your solution comprehensively after the deadline. Task 1A is worth 1 mark, the rest are worth 0.5 marks each.

Challenge 2 – Negative Implications

We have seen that implication can be re-written into an equivalent formula using the following equivalence

$$F \Rightarrow G \;\equiv\; \neg(F \wedge \neg G) \tag{2.1}$$

In this challenge we will generalise this result to rewrite all of the connectives we have seen using $\wedge$ and $\neg$. To this effect, we will show that $\{\wedge, \neg\}$ is functionally complete as we can represent all formulas using only $\wedge$ and $\neg$.

Task 2A. Using the equivalence defined in (2.1), re-write the following formula to remove all instances of the $\Rightarrow$ connective. You **must not** perform any other transformations.

$$(\neg P \Rightarrow (\neg Q \wedge Q \Rightarrow \mathbf{f})) \;\Rightarrow\; (((P \Rightarrow \neg(R \Rightarrow Q)) \wedge \neg P) \;\Rightarrow\; \neg(P \Rightarrow \neg(R \Rightarrow Q))) \tag{2.2}$$

Task 2B. The formula (2.2) can be simplified. Using **only** the equivalences (2.1) and (2.3)–(2.5) you can simplify your answer for Task 2A. Provide the most simplified formula using (2.1) and (2.3)–(2.5), with **no** instances of $\Rightarrow$. This should contain the smallest number of connectives possible.

$$F \wedge G \;\equiv\; G \wedge F \tag{2.3}$$

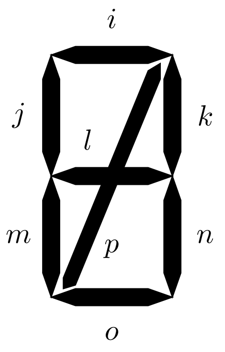
$$\neg F \wedge (F \Rightarrow G) \;\equiv\; \neg F \tag{2.4}$$

$$\neg\neg F \;\equiv\; F \tag{2.5}$$

Task 2C. Generalising the re-writing rule (2.1), we can re-write all other connectives using only $\wedge$ and $\neg$. Write a Haskell function that can re-write *any* formula into an equivalent formula that uses only $\wedge$, $\neg$, and any variables. Your function should **not** produce any double-negatives, such as $\neg\neg P$.

Submission and Marking: Your answer should be submitted on Grok. You will find the submission format explained there. You will receive some feedback from some elementary tests. These merely check that your input has the correct format; they should not be relied upon for correctness. We will test your solution comprehensively after the deadline. Task 2A and Task 2B are worth 0.5 marks each for a correct answer; Task 2C is worth 1 mark based proportionally on the number of passed test cases.

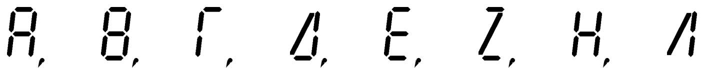
Challenge 3 – Logic on Display



In this challenge we will consider an unconventional 8-segment display which is like a 7-segment display, but has an additional diagonal LED from the top-right to bottom-left of the display. Arrays of such displays are commonly used to display characters in remote controls, blood pressure monitors, dishwashers, and other devices. We label each LED i–p, with p being the diagonal segment, as shown here.

Each LED can be on or off, but in most applications, only a small number of on/off combinations are of interest (such as the ten combinations that allow the display of a digit in the range 0–9). In that case, the display can be controlled through a small number of input wires with four wires providing 2^4 input combinations, enough to cover the ten different digits.

Here we are interested in using an 8-segment display for some Greek letters. We want it to be able to show eight different letters, namely A, B, Γ, Δ, E, Z, H, and Λ. For example, to show A, all the display segments, except o and p, should be lit up, giving the pattern . In detail, we want the eight different letters displayed respectively as:



Since there are eight letters, we need three input wires, modelled as propositional variables P, Q, and R. We will need to decide on a suitable encoding of the eight letters. One possibility encoding of the eight letters is to let A correspond to input 000 (that is, $P = Q = R = \mathbf{f}$), B to 001 (that is, $P = Q = \mathbf{f}$ and $R = \mathbf{t}$), *etc.* If we do that, we can summarise the behaviour of each input combination in the table below:

letter	P	Q	R	i	j	k	l	m	n	o	p	display
A	0	0	0	1	1	1	1	1	1	0	0	
B	0	0	1	1	1	1	1	1	1	1	0	
Γ	0	1	0	1	1	0	0	1	0	0	0	
Δ	0	1	1	0	0	1	0	0	1	1	1	
E	1	0	0	1	1	0	1	1	0	1	0	
Z	1	0	1	1	0	0	0	0	0	1	1	
H	1	1	0	0	1	1	1	1	1	0	0	
Λ	1	1	1	0	0	1	0	0	1	0	1	

Each of the eight segments i–p can be considered a propositional function of the variables P, Q, and R. This kind of display can be physically implemented with logic circuitry, using circuits to implement a Boolean function for each of the outputs. Here we assume that only three types of logic gates are available: An *and-gate* takes two inputs and produces, as output, the conjunction ($\wedge$) of the inputs. Similarly, an *or-gate* implements disjunction ($\vee$). Finally, an *inverter* takes a single input and negates it ($\neg$). We can specify such a circuit by writing down the Boolean equations for each of the outputs i–p. For example, segment i is turned off (is false) when the input is 011, 110, or, 111. So, i can be captured as $(P \vee \neg Q \vee \neg R) \wedge (\neg P \vee \neg Q \vee R) \wedge (\neg P \vee \neg Q \vee \neg R)$.

For efficiency reasons, we often want the circuit to *use as few gates as possible.* For example, the above equation for i shows that we can implement this output using fifteen gates. But $i = \neg(\neg P \wedge Q \wedge R) \wedge \neg(P \wedge Q \wedge \neg R) \wedge \neg(P \wedge Q \wedge R)$ is an equivalent implementation, using fewer gates. Moreover, the eight functions might be able to *share some circuitry.* For example, if we have already implemented a sub-circuit defined by $u = Q \wedge R$ (introducing u as a name for the sub-circuit), then we can define $i = \neg(\neg P \wedge u) \wedge \neg(P \wedge Q \wedge \neg R) \wedge \neg(P \wedge u)$, and we may be able to re-use u while implementing the other outputs (rather than duplicating the same gates). In some cases, it may even be feasible to design a circuit that is not minimal for a given function, but provides a minimal solution when all eight functions are designed.

Task 3A. Design such a circuit, using as few gates as possible. You can define any number of sub-circuits to help you reduce the gate count (simply give each a name).

Submission and Marking: Your answer should be submitted on Grok. Submit a text file `circuit.txt` consisting of one line per definition. This file will be tested automatically, so it is important that you follow the notational conventions exactly. We write $\neg$ as `-` and $\vee$ as `+`. We write $\wedge$ as `.`, or, more simply, we just leave it out, so that concatenation of expressions denotes their conjunction. Here is an example set of equations (for a different problem):

```
# An example of a set of equations in the correct format:
i = -Q R + Q -R + P -Q -R
j = u + P (Q + R)
k = P + -(Q R)
l = u + P i
u = -P -Q
# u is an auxiliary function introduced to simplify j and l
```

Empty lines, and lines that start with '`#`', are ignored. Input variables are in upper case. Negation binds tighter than conjunction, which in turn binds tighter than disjunction. So the equation for i says that $i = (\neg Q \wedge R) \vee (Q \wedge \neg R) \vee (P \wedge \neg Q \wedge \neg R)$. Note the use of a helper function u, allowing j and l to share some circuitry. Also note that we do not allow any feedback loops in the circuit. In the example above, l depends on i, so i is not allowed to depend, directly or indirectly, on l (and indeed it does not).

To test your equations and count the number of gates used, you can click ***Terminal*** and enter the command `test`. To stop the ***Terminal***, click ***Stop***.

There is one mark for a correct solution. An additional 0.5 is awarded if a correct solution uses 26 gates or fewer. A further 0.5 is awarded if a correct solution uses 20 gates or fewer.

Optionally, you can submit your circuit design to a leaderboard. Your position on this board is not reflective of your final grade and can be used anonymously. The leaderboard site can be found here `https://comp30020.ddns.net/leaderboard`.

Challenge 4 – Property-Based Testing

Unlike unit tests that only test a single use case of a program, Property-Based Testing allows programmers to provide a specification of their programs, as logical properties that should hold if their program is implemented correctly. There are two types of properties that one can test about their code. One is data invariants which are light sanity checks and the others are full functional specifications.

For example, consider the following definition of `reverse` in Haskell

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

A nice property of a `reverse` function is that when composed with itself, it is the identity. That is, to say

$$\forall a \; \texttt{reverse}\ (\texttt{reverse}\ a) \;=\; a \tag{4.1}$$

This property is not enough to show that the `reverse` is functionally correct, so we say that this is a data invariant of `reverse`. For example, consider we replaced the definition of `reverse` with `id` (the identity function), the property (4.1) still holds.

For a full specification of the `reverse` function, we require a stronger property as follows

$$\begin{aligned} &\forall a \; \texttt{length}\ (\texttt{reverse}\ a) \;=\; \texttt{length}\ a \\ &\quad \wedge \;\; \forall i \; 0 \le i < \texttt{length}\ a \;\Rightarrow\; \texttt{reverse}\ a \;\texttt{!!}\; i \;=\; a \;\texttt{!!}\; (\texttt{length}\ a - i - 1) \end{aligned} \tag{4.2}$$

This specifies that the length of the reversal of some list, a, is the same as the length of a, and for each value of the reversal of some list, a, at index i, the value in a at the opposing end of a must be the same as said value in the reversal of a. Take a moment to convince yourself this is true for a correct implementation of `reverse` and some example lists a.

In Haskell, the *QuickCheck* library[1] provides property-based testing. This library is probably older than many of you, appearing first in 1999, and has been ported to well over 30 languages. In summary, QuickCheck generates a series of randomised inputs that are tested against the properties similar to the properties we have previously seen. QuickCheck then reports any test cases that fail as *counter-examples*. **You do not need to learn or understand QuickCheck to solve this challenge.**

Through this challenge, we will use a simplified model of property-based testing in Haskell. We are concerned with the functional correctness of *sorting functions*. To begin, we will be considering the following *merge sort*, `msort`.

```
msort :: (Ord a) => [a] -> [a]
msort xs@(_:_:_) = msort (take n xs) `merge` msort (drop n xs)
  where n = length xs `div` 2
        merge [] rs = rs
        merge ls [] = ls
        merge lls@(l:ls) rrs@(r:rs)
            | l < r     = l:merge ls  rrs
            | otherwise = r:merge lls rs
msort xs = xs
```

As we may have multiple sorting functions to test, the functions we will implement to test for these properties we will see will parameterise which sorting function is to be tested, and so will have the following form

[1]`https://hackage.haskell.org/package/QuickCheck`

```
sortProperty :: (Ord a) => ([a] -> [a]) -> [a] -> Bool
sortProperty sort input = {- Boolean expression -}
```

Testing individual sorting functions is then performed with `sortProperty msort`, *etc.*

Task 4A. Implement a function, `sortLength`, that checks the following property: *the result of sorting some input must have the same length as the input.*

Task 4B. Implement a function, `sortHead`, that checks the following property: *if the input is not empty, then the head element of the sorted input is the least element of the input.*

Task 4C. Implement a function, `sortIsSorted`, that checks the following property: *the result of sorting some input is in sort-order, i.e., the result is a non-decreasing list.*

Task 4D. The following is a *functional specification* of all sorting functions:

```
sortSpec :: (Ord a) => ([a] -> [a]) -> [a] -> Bool
sortSpec sort input =
    elem (sort input) (permutations input) && sortIsSorted sort input
```

That is to say, a function sorts its input if the output is a permutation of the input and the output is in sort-order.

With the following (incorrect) implementation of `qsort`, provide two values, `example` and `counterExample`, that are an example and a counter-example, respectively, for the `sortSpec` functional specification with respect to the `qsort` function.

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (pivot:rest) = qsort lesser ++ [pivot] ++ qsort greater
  where lesser  = filter (< pivot) rest
        greater = filter (> pivot) rest
```

That is, `sortSpec qsort example` should be `True` and `sortSpec qsort counterExample` should be `False`.

Submission and Marking: Your answer should be submitted on Grok. You will find the submission format explained there. You will receive some feedback from some elementary tests. These merely check that your input has the correct format; they should not be relied upon for correctness. We will test your solution comprehensively after the deadline. Your functions should hold only for the properties asked. Each of Task 4A, Task 4B, and Task 4C are worth 0.5 marks, each based proportionally on the number of test cases passed. Task 4D is worth 0.5 marks, with 0.25 marks awarded for each correct value provided.

[2] *Note:* this specification depends on the definition of the property from Task 4C that you must implement yourself.

Challenge 5 – Interpreting Resolutions

Consider the following predicate logic formulas.

$$\begin{aligned} F: &\quad (\forall x\ Q(x)) \lor \exists x((\forall y\ R(y,x) \lor Q(x)) \Rightarrow \exists z \forall y\ P(z,y)) \\ G: &\quad \exists x \forall y\ (P(x,y) \lor (\exists z\ R(y,z) \Rightarrow \forall w\ Q(w))) \end{aligned}$$

Task 5A. Show that F is non-valid, by providing an appropriate interpretation $\mathcal{I}$.

Task 5B. Show that $F \lor \neg G$ is valid using resolution, explicitly stating all substitutions used.

Submission and Marking: Your answers to Challenge 5 and Challenge 6 should be submitted through Gradescope as *a single PDF document, no more than 2 MB in size.* Marks are primarily allocated for correctness, but elegance and how clearly you communicate your thinking will also be taken into account. The process of resolution should be displayed as a tree.

Challenge 6 – Evenness

The notation we use for first-order predicate logic includes function symbols. This allows a very simple representation of the natural numbers. Namely, for natural numbers, we use terms built from a constant symbol (here we choose a, but any other symbol would do) and a one-place function symbol (we will use s, for "successor"). The idea is that 0 is represented by a, 1 by $s(a)$, 2 by $s(s(a))$, and so on. In general, $s(x)$ represents the successor of x, that is, $x+1$. Logicians prefer this "successor" notation, because it uses so few symbols and supports recursive definition-—a natural number is either 'a' (the base case), or it is of the form '$s(x)$', where x is a term representing a natural number. (Of course, for practical use, we prefer the positional decimal system.)

With successor notation, we can capture addition by introducing a predicate symbol for the addition relation, letting $P(x, y, z)$ stand for $x + y = z$:

$$\forall x\ P(a, x, x) \qquad \text{(Identity element)} \qquad (6.1)$$

$$\forall x \forall y \forall z\ (P(x, y, z) \Rightarrow P(s(x), y, s(z))) \qquad \text{(Recursive relation)} \qquad (6.2)$$

$$\forall x \forall y\ (P(x, y, z) \Rightarrow P(y, x, z)) \qquad \text{(Commutativity)} \qquad (6.3)$$

Similarly, using the addition relation we can now define the evenness of a number by introducing the predicate symbol for evenness, letting $E(x)$ stand for x *is even*:

$$\forall x \exists y\ (E(x) \Rightarrow P(y, y, x)) \qquad (6.4)$$

$$\forall x \forall y\ (P(y, y, x) \Rightarrow E(x)) \qquad (6.5)$$

Task 6A. Now, the goal is to prove the well-known property of natural numbers that if n is an even number then $n + 2$ is also even. Use resolution to show that

$$\forall x\ (E(x) \Rightarrow E(s(s(x)))) \qquad (6.6)$$

is a logical consequence of the axioms (6.1)–(6.5).

Task 6B. We have defined what an even number is and a theorem about even numbers, but we still don't know if even numbers exist! Using resolution, show that

$$\exists x\ E(s(s(s(x))))$$

is a logical consequence of the axioms (6.1)–(6.5) and the theorem (6.6). The resolution proof provides a sequence of most general unifiers, one per resolution step, and when these are composed in the order they were generated, you have a substitution that solves the constraint $E(s(s(s(x))))$. Give that substitution and explain what it means in terms of natural numbers.

Submission and Marking: Your answers to Challenge 5 and Challenge 6 should be submitted through Gradescope as *a single PDF document, no more than 2 MB in size*. Marks are primarily allocated for correctness, but elegance and how clearly you communicate your thinking will also be taken into account. The process of resolution should be displayed as a tree.

Further Submission Advice

The deadline is 16 September at 23:00. Late submission will be possible, but a late submission penalty will apply: a flagfall of 1 mark, and then 1 mark per 12 hours late.

Note that on Grok, "saving" your file does not mean submitting it for marking. To submit, you need to click Mark and then Submit. You can submit as many times as you like. What gets marked is the last submission you have made before the deadline.

For Challenge 5 and Challenge 6, if you produce an MS Word document, it must be exported and submitted as PDF, and satisfy the space limit of 2 MB. We also accept neat hand-written submissions, but these must be scanned and provided as PDF. Illegible or poorly-written submission will likely attract few, if any, marks. If you scan your document, make sure you set the resolution so that the generated document is no more than 2 MB. The Canvas module, from which you downloaded this document, has advice to help you satisfy the 2 MB requirement while maintaining readability.

Being neat is easier if you type-set your answers, but not all typesetting software does a good job of presenting mathematical formulas. The best tool is LaTeX, which is worth learning if you expect that you will later have to produce large technical documents. Admittedly, diagrams are tedious to do with LaTeX, even when using sophisticated packages such as Ti*k*Z. You could, of course, mix typeset text with hand-drawn diagrams. In case you want to use this assignment to get some LaTeX practice, we will leave the source for this document in the Canvas module where you find the PDF version.

Make sure that you have enough time towards the end of the assignment to present your solutions carefully. A nice presentation is sometimes more time consuming than solving the problems. Start early; note that time you put in early usually turns out more productive than a last-minute effort. For Challenge 3 in particular, you don't want to submit some "improved" solution a few minutes before the deadline, as it may turn out to be wrong and you won't have time to change your mind.

Academic Honesty Statement

By submitting work for assessment you implicitly declare that you understand the University's policy on ***academic integrity*** and that the work submitted is your original work. You declare that you have not been unduly assisted by any other person (collusion). In this assignment, ***individual work*** is called for, but ***if you get stuck***, you can use the Ed Discussion board to ask any questions you have. As long as nobody directly gives away solutions, our discussion forum is both useful and appropriate for this; we can all use it to support each other's learning. If your question is simply to clarify some aspect of the assignment, your post can be 'public', but if it reveals anything about your attempted solution, make sure it is submitted as a 'private' post to the teaching team. Soliciting help from sources other than the above will be considered cheating and will lead to disciplinary action.