- Remember to register your attendance using the UoB Check-In app. Either

  1. download, install, and use the native app[a] available for Android and iOS, or
  2. directly use the web-based app available at

     https://check-in.bristol.ac.uk

  noting the latter is also linked to via the `Attendance` menu item on the left-hand side of the Blackboard-based unit web-site.

- The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. If you encounter a problem (e.g., a lab. workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: you can contact them, to report then resolve said problem, via

  https://www.bristol.ac.uk/it-support

- The lab. worksheet is written *assuming* you work in the lab. using UoB-managed and thus *supported* equipment. If you need or prefer to use your own equipment, however, various *unsupported*[b] alternatives available: for example, *you* could 1) manually install any software dependencies yourself, *or* 2) use the unit-specific Vagrant[c] box by following instructions at

  https://cs-uob.github.io/COMS10015/vm

- The questions are roughly classified as either C (for core questions, that *should* be attempted within the lab. slot), A (for additional questions, that *could* be attempted within the lab. slot), or R (for revision questions). Keep in mind that we only *expect* you to attempt the C-class questions: the other classes are provided *purely* for your benefit and/or interest, so there is no problem with nor penalty for totally ignoring them.

- There is an associated set of solutions is available, at least for the C-class questions. These solutions are there for you to learn from (e.g., to provide an explanation or hint, or illustrate a range of different solutions and/or trade-offs), rather than (purely) to judge your solution against; they often present *a* solution vs. *the* solution, meaning there might be many valid approaches to and solutions for a question.

- Keep in mind that various mechanisms exist to get support with and/or feedback on your work; these include both in-person (e.g., the lab. slot itself) *and* online (e.g., the unit forum, accessible via the unit web-site) instances.

[a] https://www.bristol.ac.uk/students/support/it/software-and-online-resources/registering-attendance

[b] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.

[c] https://www.vagrantup.com

COMS10015 lab. worksheet #1: Boolean algebra

§1. C-class, or core questions

▷ **Q1[C].** This lab. worksheet makes use[1] of SymPy, a tool (in fact a Python-based library) for symbolic computation and, e.g., Boolean algebra: doing so enables hands-on exploration of associated concepts, and thus aims to enhance your understanding based on the lecture slot(s).

A brief introduction. The goal[2] of *this* question is to explain the high-level features in and workflow for using this tool, offering experience that will allow you to engage with tasks and challenges presented in *later* questions. Start by opening the SymPy Live shell

https://www.sympy.org/en/shell.html

in a web-browser. You *should* eventually see a something similar to Figure 1, which has been annotated to highlight several major features. Since SymPy is a Python library, the SymPy Live shell is really just a Python shell with SymPy made available: a Read-Eval-Print Loop (REPL)[3] style of user interface therefore supports interactive evaluation of Python statements and expressions. Within the user interface,

- the input pane is where input expressions and statements are buffered ready for be evaluation,
- the output pane is where the result of evaluating the input buffer is rendered, and
- the control pane includes various buttons, which, e.g., allow the input buffer to be cleared or evaluated.

As the name suggests, the field of symbolic computation focuses on the computation of mathematical objects, e.g., expressions and functions, in a symbolic manner: doing so implies variables are not given concrete values, so are represented as symbols. SymPy supports general-purpose symbolic computation, but, as a subset, can deal with *Boolean* expressions and functions.

Some initial, exploratory tasks.

a. Figure 2 compares various styles of syntax (or notation) for Boolean operators, each of which allows definition of Boolean expressions and functions but are useful and so used in different contexts. Towards the left-hand side, we see the Mathematics-oriented syntax used, e.g., in the lecture slot(s). Towards the right-hand side, we see two styles of programming-oriented syntax used, e.g., in Python or C. These right-hand columns capture both

   - *decisional* operators, e.g., && : $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$, and
   - *computational* operators, e.g., & : $\mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}^n$,

   which are (subtly) different: we typically use the former within a condition and the latter within an assignment, and will focus here on the later.

   Type (or copy-and-paste) the following statements into the SymPy input pane, then press return to have them evaluated:

```
a, b, c = symbols( 'a b c', boolean = True )

f = ~( a | b ) | c
g = b & ( a | c )
```

   Step-by-step, this acts to 1) introduce some SymPy symbols, i.e., `a` to `c` to represent the variables a to c, then 2) define some Boolean functions, i.e., `f` and `g`, using `&`, `|`, and `~` to denote AND, OR, and NOT respectively: doing so thereby produces SymPy representations of the Boolean functions $f(a,b,c,) = \neg(a \vee b) \vee c$ and $g(a,b,c,) = b \wedge (a \vee c)$. Now, one-by-one, type (or copy-and-paste) the statements

```
print( f )
```

   and

[1] See, e.g., https://sympy.org noting that although we focus specifically on SymPy 1.5.1 executed on Python 3.x, use of *other* versions is plausible modulo any minor incompatibilities.

[2] The high-level focus implies that a *non*-goal is therefore an in-depth tutorial-style introduction. Rather, there is an emphasis on *you* to, e.g., explore and use the wider documentation as and when need be: see, e.g., https://docs.sympy.org

[3] https://en.wikipedia.org/wiki/Read-eval-print_loop

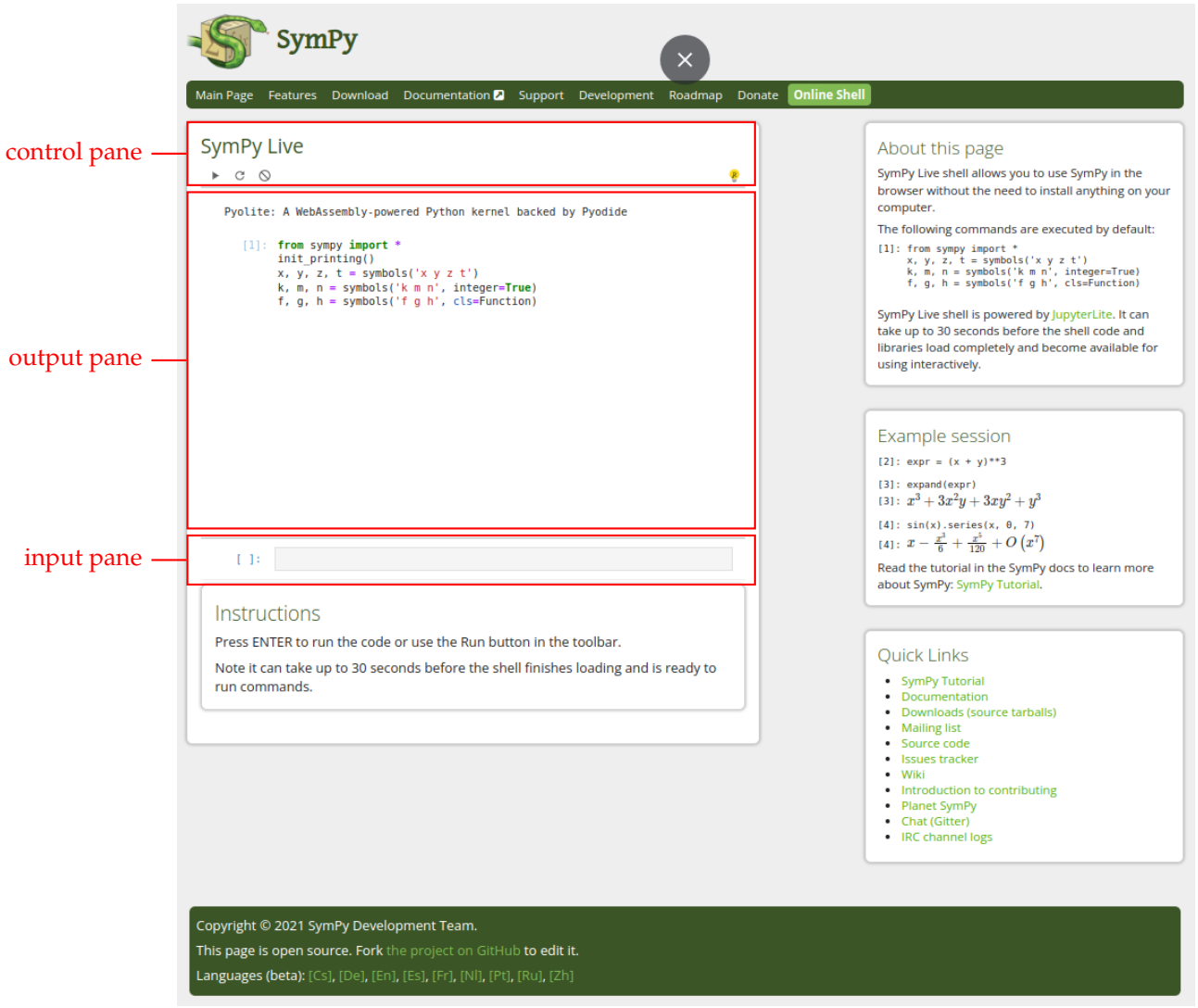


Figure 1: *The online SymPy shell, annotated to illustrate major features.*

r is x	$\equiv$	$r = x$	$\cong$	`r == x`	$\cong$	`r = x`
r is NOT x	$\equiv$	$r = \neg x$	$\cong$	`r != x`	$\cong$	`r = ~x`
r is x NAND y	$\equiv$	$r = x \overline{\wedge} y$	$\cong$	`r != x && y`	$\cong$	`r = ~( x & y )`
r is x NOR y	$\equiv$	$r = x \overline{\vee} y$	$\cong$	`r != x || y`	$\cong$	`r = ~( x | y )`
r is x AND y	$\equiv$	$r = x \wedge y$	$\cong$	`r == x && y`	$\cong$	`r = x & y`
r is x OR y	$\equiv$	$r = x \vee y$	$\cong$	`r == x || y`	$\cong$	`r = x | y`
r is x XOR y	$\equiv$	$r = x \oplus y$			$\cong$	`r = x ^ y`

Figure 2: *Comparison between syntactic forms for Boolean algebra.*

```
f
```

into the SymPy input pane, then press return to have them evaluated. Both render `f` in the output pane: the first does so in ASCII, whereas the second does so in pretty-printed[4] LaTeX which therefore matches the lecture slot(s). Although both approaches are useful, for consistency we stick with the first i.e., explicit use of `print`, throughout what follows.

b. Although SymPy represents `f` symbolically, we can have it evaluate the expression by providing a concrete value for each symbol involved.

Type (or copy-and-paste) the following statements into the SymPy input pane, then press return to have them evaluated:

```
print( f.subs( { a : True, b : False, c : True } ) )
```

By assigning values to `a`, `b`, and `c`, and then substituting these into `f`, we can then evaluate the function. The rendered output, i.e., **true**, shows that doing so is equivalent to manually reasoning as follows:

$$\begin{aligned} a &= \textbf{true} \\ b &= \textbf{false} \\ c &= \textbf{true} \end{aligned}$$

$$\begin{aligned} f &= \neg(a \vee b) \vee c \\ &= \neg(\textbf{true} \vee \textbf{false}) \vee \textbf{true} \\ &= \textbf{true} \end{aligned}$$

c. Notice that `f` is not in Sum of Products (SoP) (or disjunctive normal) form *or* Product of Sums (PoS) (or conjunctive normal) form. We can ask SymPy to manipulate it, however, into such a form.

Type (or copy-and-paste) the following statements into the SymPy input pane, then press return to have them evaluated:

```
from sympy.logic.boolalg import *

print( to_dnf( f ) )
print( to_cnf( f ) )
```

The rendered output, i.e., $c \vee (\neg a \wedge \neg b)$ and $(c \vee \neg a) \wedge (c \vee \neg b)$, *is* now in SoP and PoS form respectively.

▷ **Q2[C].** Consider the following, similar questions:

- simplify the Boolean expression
$$(a \wedge b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge b \wedge \neg c)$$
into a form that contains the fewest operators possible,
- simplify the Boolean expression
$$(\neg(a \vee b) \wedge \neg(c \vee d \vee e)) \vee \neg(a \vee b)$$
into a form that contains the fewest operators possible.

For each question, try to
a. produce a *manual* solution,
b. research and apply features in SymPy to produce an *automated* solution,
c. compare the two solutions, i.e., verify the correctness of your manual solution using the automatic solution as a reference, and explain any differences you observe.

§2. R-class, or revision questions

▷ **Q3[R].** There is a (large) set of questions available at

https://assets.phoo.org/COMS10015_2025_TB-4/csdsp/sheet/misc-revision_q.pdf

There are far too many for the lab. slot(s) alone, but, in the longer term, the idea is simple: attempt to answer the questions, applying theory covered in the lecture(s) to do so, as a means of revising and thereby *ensuring* you understand the material. Note that the set of questions is general-purpose, in the sense it covers *all* topics covered by the lecture(s): you will need to be selective, and only consider questions related to this lab. slot.

[4] https://en.wikipedia.org/wiki/Prettyprint

§3. A-class, or additional questions

▷ **Q4[A].** Consider the following question: given

$$\begin{aligned} f(a,b) &= (b \wedge \neg a) \vee (a \wedge \neg b) \\ g(a,b) &= (a \vee b) \wedge \neg(a \wedge b) \end{aligned}$$

decide whether or not $f \equiv g$. Try to

a. produce a *manual* solution,

b. research the concept of Boolean satisfiability[5] (or SAT),

c. use SymPy to apply this concept, and thereby produce an *automated* solution.

▷ **Q5[A].** bOOleO is a card game involving Boolean algebra. Using AND(r), OR(r) and XOR(r) to denote cards for Boolean AND, OR and XOR operators whose output is r, a full deck of bOOleO cards includes the following:

- 48 Boolean operator cards, namely
  - $8 \times$ AND(0) cards,
  - $8 \times$ AND(1) cards,
  - $8 \times$ OR(0) cards,
  - $8 \times$ OR(1) cards,
  - $8 \times$ XOR(0) cards,
  - $8 \times$ XOR(1) cards,
  - $8 \times$ NOT cards

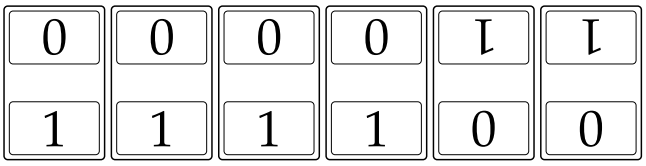
  and
- 6 Boolean value cards (which have a 0 and 1 on).

There is a Wikipedia entry

https://en.wikipedia.org/wiki/Booleo

with a brief overview of the rules, but, for completeness, an alternative follows:

- Decide which player will act as the dealer: they should shuffle the operator cards, and (separately) the value cards. The dealer should then place the value cards between the players (st. the short edges face the players). An example might be as follows:



  The top player faces downward at the value cards (read from left-to right) 110000; the bottom player faces upward at the value cards 111100.

  Finally, the dealer should deal 4 operator cards to each player: this becomes their hand. All remaining operator cards form a draw deck from which players take cards (the draw deck is placed face down, meaning the card types cannot be seen).
- The goal of the game, for each player, is to form a valid tree (or pyramid) of operator cards. Some rules for forming the tree are as follows:
  - For a given player, the tree should start from (i.e., the base should be) the value cards and extend outward towards them.
  - The tree should end with (i.e., the tip should be) a single output matching the right-most value card (in the example, for the bottom player the output should be 0).

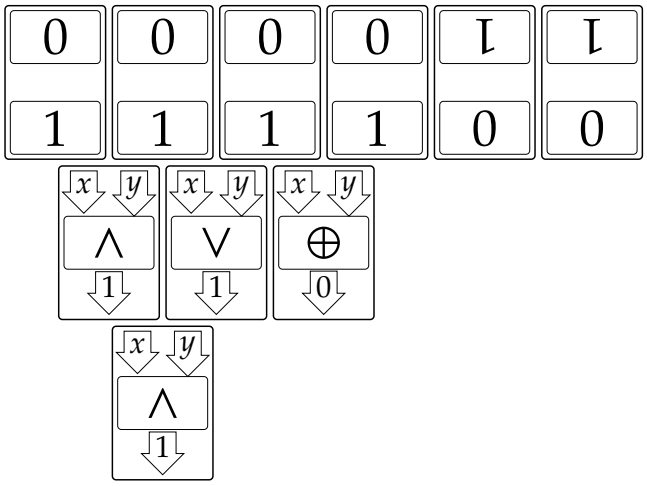
[5] See, e.g., https://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

  - The inputs and outputs of operator cards in the tree should respect Boolean algebra: if one evaluates the operator (by using inputs from the layer of tree above it), the output should match the following:

x	y	valid operator card		
0	0	AND(0)	OR(0)	XOR(0)
0	1	AND(0)	OR(1)	XOR(1)
1	0	AND(0)	OR(1)	XOR(1)
1	1	AND(1)	OR(1)	XOR(0)

- The game is turn-based, st. one player has a turn then the other and so on until the game terminates. The non-dealer should start. Within their turn, a player
  - draws a new operator card from the draw deck, then
  - plays *or* discards an operator card from their hand

  meaning that after their turn, they *still* have a 4 card hand. If the player plays a card, this implies they will either
  - use a NOT operator card, which flips a value card (literally rotating it) selected by the player,
  - place any other operator card in the tree, thus extending it, or
  - replace any operator card in the tree.
- Some special-cases need further explanation:
  - When a card needs to be discarded, it is placed on a discard deck next to the draw deck. If/when the draw deck is empty, the dealer should shuffle the discard deck which then becomes the new draw deck.
  - When a turn is complete, one or *both* trees need to be reevaluated: any part of either tree which is invalid (e.g., the output of a given operator card no longer matches the inputs), must be discarded. The reevaluation process should start at the value cards, and progress layer-by-layer through the entire tree.
  - An operator card which is left "dangling" with no input(s), e.g., due to an operator card in the layer above having been discarded, is *not* viewed as invalid per se: rather, it is left inactive and then reevaluated when an input becomes available.
  - When a NOT operator card is played, it is removed from play permanently rather than being discarded.
  - When a NOT operator card is played, the opponent can optionally *cancel* the flipping effect by playing a NOT of their own during their turn. The need to wait and see what the opponent does suggests you should wait to see if the tree(s) need reevaluation, and only do so if the value card is flipped after all.
- Ignoring the top player, imagine the game state wrt. the bottom player is as follows:



  Assuming the relevant cards are available, some valid moves and their implication then include:
  - Extend the tree by placing an operator card next to the lower AND(1) card, e.g., XOR(1) or AND(0).
  - Update the tree by replacing the OR(1) operator card, for example: if we replace it with AND(1) then the card below it is still valid so is retained, if we replace it with XOR(0) (which is still valid wrt. the inputs) then the card below becomes invalid and is discarded.

– Use a NOT operator, on the right-most value card whose value is 1 for example. In this case it will flip from 1 to 0, so the XOR(0) card below becomes invalid and is discarded.

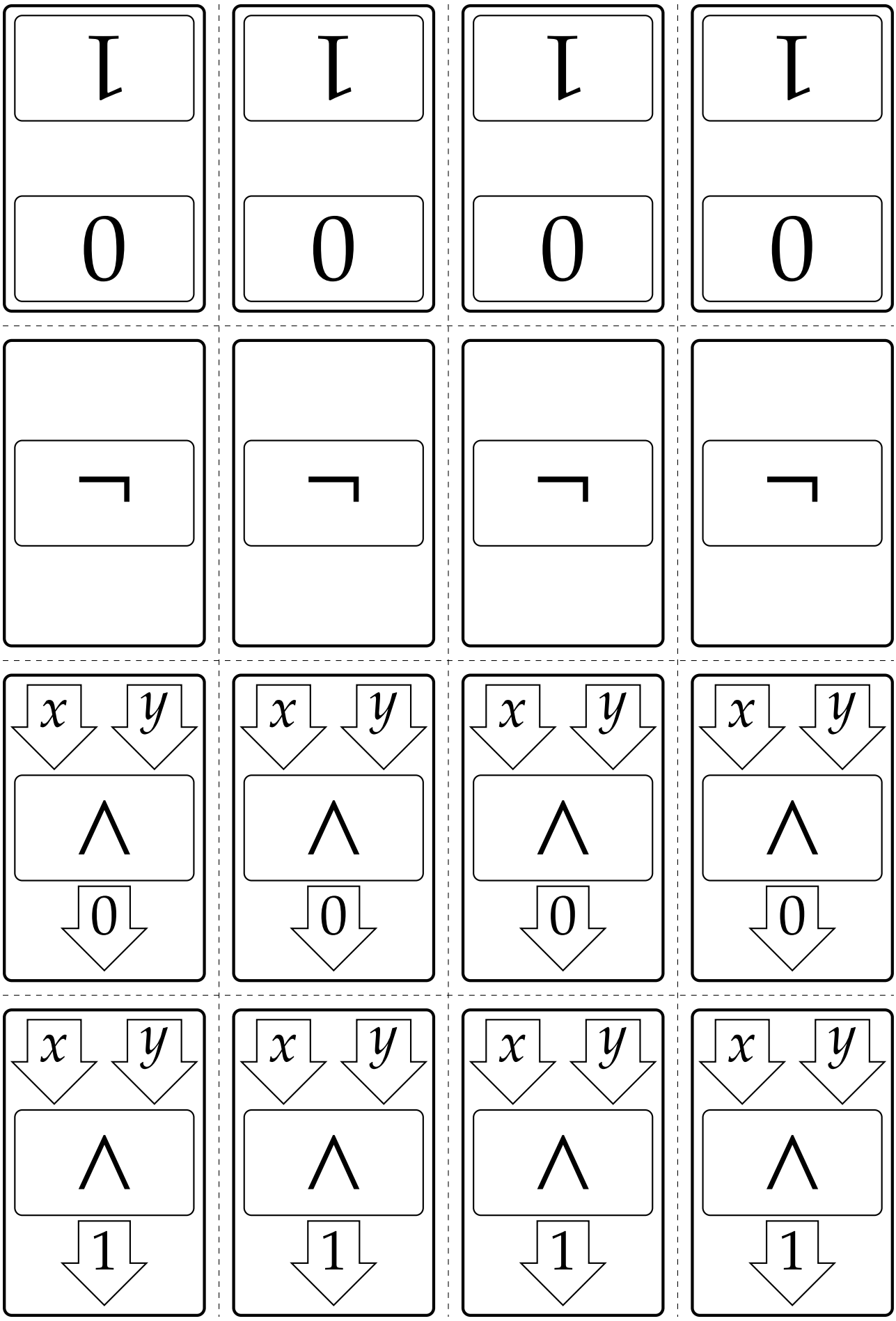
Based on this, have a go at the following tasks:

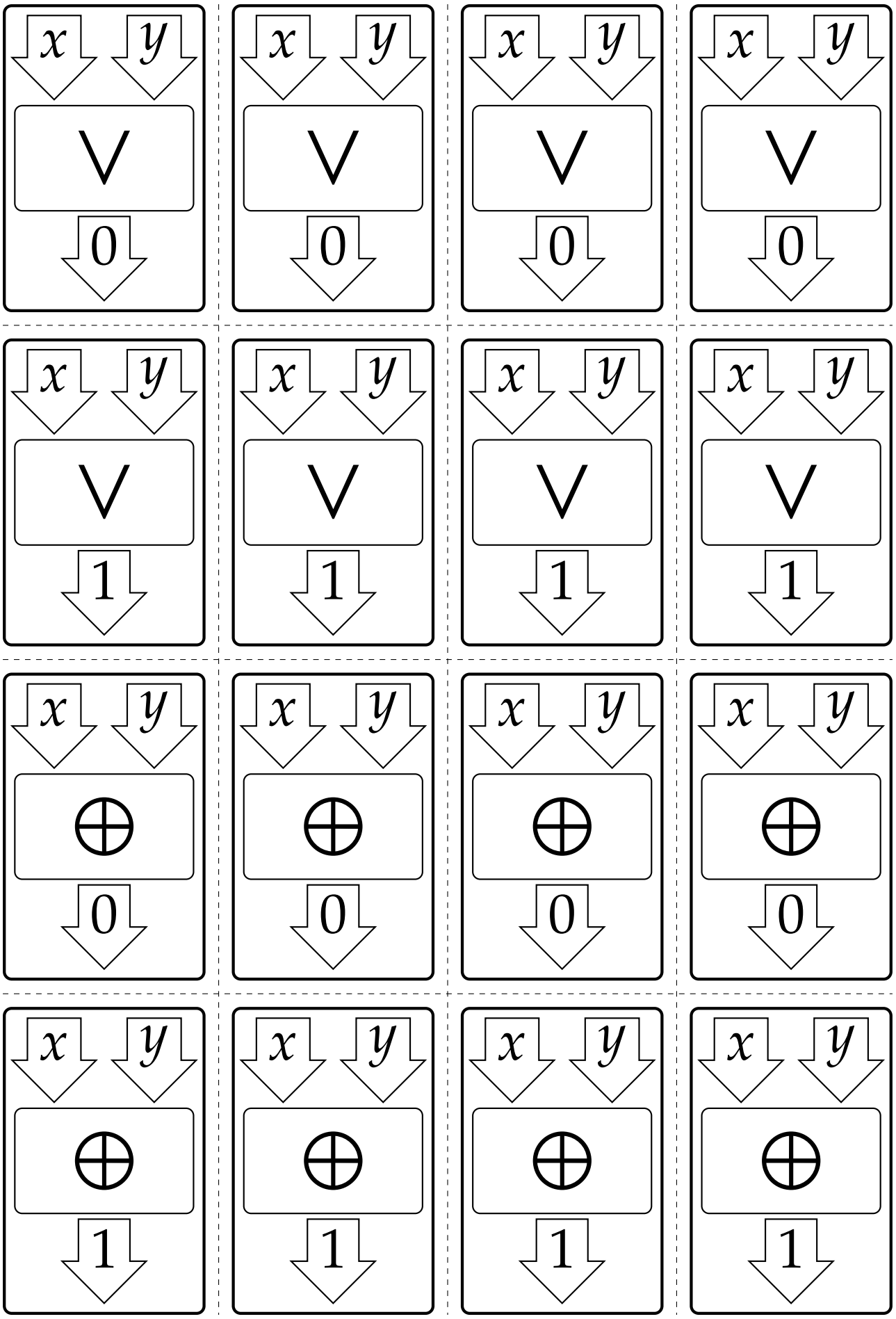
a. The pages toward the end of this worksheet include enough cards for a *half* deck: this implies you first need to find an opponent to play against, and then combine your cards. Find another student, and see if you can complete a game of bOOleO.

b. Based on your experience, think about the following:

- There are various strategies relating to different aspects of the game. Think about the first row of operator cards placed below the value cards: is there a reason to favour one type over another, and why is this the case?
- It is possible for one or other player to hit a "dead end" meaning the game cannot terminate (which is quite annoying): can you identify this case, and suggest a way to resolve the problem?
- Various extensions or variations of bOOleO are possible; bOOleO-N, for example, introduces NAND and NOR operator cards. Can you think of any other interesting variations?

  For example, one *could* imagine a "wildcard" operator of some sort: is there a Boolean or physical justification for such an operator? Or, what happens if you add n-input, m-output operators (for $n > 2$ and/or $m > 1$): can you think of any interesting examples? Or, what about involving more than two players?


⊥
0
¬
x
y
∧
1


x y
∨
0
x y
∨
0
x y
∨
0
x y
∨
0
x y
∨
1
x y
∨
1
x y
∨
1
x y
∨
1
x y
⊕
0
x y
⊕
0
x y
⊕
0
x y
⊕
0
x y
⊕
1
x y
⊕
1
x y
⊕
1
x y
⊕
1