Introduction to logic programming

Dr. Giuseppe Maggiore

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Opening

- Consider many primitive and “obvious” concepts
- Who in their right mind would feel the need to “explain” natural numbers, addition, etc.?
- Similarly variables, `if`, `while`, functions, etc. are often given for granted

Introduction

Opening

- In this course we challenge the assumption of *primitive concepts*
- We will start with some bare-bones logical operators
- We will use an implementation of these operators, so everything we do *can actually be run on a computer*)
- We will use this language to define, from scratch, multiple basic concepts

Introduction

Opening

- The result is that we will rediscover how basic concepts are actually not basic
- We will be able to recompose them in new forms
- For example in terms of new programming languages

Introduction

During this course we will:

- introduce informal logical reasoning
- introduce our logic programming language and its internal mechanisms;
- give a first example of logic in action to define basic boolean expressions
- use logic to define *apparently atomic* and unrelated concepts such as natural numbers
- construct complex data structures such as lists, binary search trees, and even balanced binary search trees
- build a small programming language
- conclude with a short and woefully incomplete presentation of fragments of computability

Introduction

Lecture agenda

- Introduce logical reasoning
- Introduce an intuition of logical processes
- Show a first example of using logic to define computations

Informal introduction

Informal introduction

What is logic?

- The basic rules of logic are very simple
- Logic is just a way to transform and manipulate symbols
- These symbols are *usually* common signs such as numbers, letter, Greek letters, etc.[a]

[a]it is not forbidden to use any symbol we find useful or interesting

Informal introduction

What is logic?

- In the following we will use smileys and other icons as *symbols*
- We will then define a series of *rules* which *transform* series of symbols

Informal introduction

What is logic?

- After defining symbols and rules we can actually use a logical system to process information
- We start with a given proposition, input to the whole process
- A *proposition* is just a series of symbols, such as for example: `3 + 2 = 5 or` ☺ ☺ 🕯 🕯
- We keep applying rules to proposition until we reach the final answer

Informal introduction

What is logic?

- Let us consider a full example. Consider the symbols of our language to be:
  - A smiley ☺
  - A candle 🕯
  - A tree 🌳
  - A coffee cup ☕

Informal introduction

What is logic?

- We consider a proposition to be true if and only if we can process it until we reach a ☕. [a]
- Our rules are
  - **(G0)** A ☕ means we are done
  - **(R1)** Two ☺ followed by a 🌳, and then further followed by `r`, means that we will have to process `r` to find the answer
  - **(R2)** Two 🕯 followed by a 🌳, and then further followed by `r`, means that we will have to process `r` to find the answer

[a] Given the extreme importance of coffee in the diet of mathematicians and computer scientists, equating coffee with truth does not really seem that illogical a step

Informal introduction

What is logic?

- Consider now the input proposition of 🙂 🙂 🌳 🕯 🕯 🌳 🙂 🙂 🌳 ☕

Informal introduction

What is logic?

- Consider now the input proposition of ☺ ☺ 🌳 🕯 🕯 🌳 ☺ ☺ 🌳 ☕
- We begin by using **R2**, therefore obtaining: 🕯 🕯 🌳 ☺ ☺ 🌳 ☕

Informal introduction

What is logic?

- Consider now the input proposition of ☺ ☺ 🌳 🕯 🕯 🌳 ☺ ☺ 🌳 ☕
- We begin by using **R2**, therefore obtaining: 🕯 🕯 🌳 ☺ ☺ 🌳 ☕
- We then use rule **R1**, therefore obtaining: ☺ ☺ 🌳 ☕

Informal introduction

What is logic?

- Consider now the input proposition of ☺ ☺ 🌳 🕯 🕯 🌳 ☺ ☺ 🌳 ☕
- We begin by using **R2**, therefore obtaining: 🕯 🕯 🌳 ☺ ☺ 🌳 ☕
- We then use rule **R1**, therefore obtaining: ☺ ☺ 🌳 ☕
- We then use rule **R2**, therefore obtaining: ☕

Informal introduction

What is logic?

- Consider now the input proposition of ☺ ☺ 🌳 🕯 🕯 🌳 ☺ ☺ 🌳 ☕
- We begin by using **R2**, therefore obtaining: 🕯 🕯 🌳 ☺ ☺ 🌳 ☕
- We then use rule **R1**, therefore obtaining: ☺ ☺ 🌳 ☕
- We then use rule **R2**, therefore obtaining: ☕
- Now according to **G0** we are done: **the proposition was true** within our logical system

Informal introduction

What is logic?

- Consider now the new input proposition of ☺ ☺ 🌳 ☺ ☺ 🌳 🕯 🌳 ☕

Informal introduction

What is logic?

- Consider now the new input proposition of ☺ ☺ 🌳 ☺ ☺ 🌳 🕯 🌳 ☕
- We begin by using **R2**, therefore obtaining: ☺ ☺ 🌳 🕯 🌳 ☕

Informal introduction

What is logic?

- Consider now the new input proposition of ☺ ☺ 🌳 ☺ ☺ 🌳 🕯 🌳 ☕
- We begin by using **R2**, therefore obtaining: ☺ ☺ 🌳 🕯 🌳 ☕
- We then use rule **R1**, therefore obtaining: 🕯 🌳 ☕

Informal introduction

What is logic?

- Consider now the new input proposition of ☺ ☺ 🌳 ☺ ☺ 🌳 🕯 🌳 ☕
- We begin by using **R2**, therefore obtaining: ☺ ☺ 🌳 🕯 🌳 ☕
- We then use rule **R1**, therefore obtaining: 🕯 🌳 ☕
- Unfortunately now we cannot apply:
  - rule **R1**, because we have no ☺ at the beginning
  - rule **R2**, because we have no 🕯 at the beginning
  - rule **G0** because there is no lonely ☕

Informal introduction

What is logic?

- Consider now the new input proposition of ☺ ☺ 🌳 ☺ ☺ 🌳 🕯 🌳 ☕
- We begin by using **R2**, therefore obtaining: ☺ ☺ 🌳 🕯 🌳 ☕
- We then use rule **R1**, therefore obtaining: 🕯 🌳 ☕
- Unfortunately now we cannot apply:
  - rule **R1**, because we have no ☺ at the beginning
  - rule **R2**, because we have no 🕯 at the beginning
  - rule **G0** because there is no lonely ☕
- the process is *stuck*: **the proposition was not true** within our logical system

Inference systems

Inference systems

Introduction

- In this section we present a more formal treatment of inference systems
- Moreover, we give a clearer syntax and semantics of symbols and rules
- We use a formalism which is also a programming language, *Meta-Casanova*
- `https://github.com/vs-team/metacompiler`

Inference systems

Syntax for symbol definitions

- Two sorts of symbols in Meta-Casanova
- `Data` symbols are used to store and structure information
- `Function` symbols are used to transform information

Inference systems

Data symbols

- Data symbols are used to assemble **propositions** (or *expressions*)
- Propositions are **nested** sequences of symbols
- **Nesting** is defined by priority or parentheses

```
Data "NAME" : TYPE
```

Inference systems

Basic definition of data

- `NAME` is a sequence of alphanumeric characters and arithmetic operators
- The new symbol will be useable whenever an expression cathegorized as `TYPE` is expected

```
Data "TRUE" : Expr
Data "FALSE" : Expr
```

Inference systems

Example: boolean truth-values

- Some data symbols can be used to assemble complex expressions
- The syntax for such a definition then becomes:
- `Data` $\texttt{P}_1$ `-> ... ->` $\texttt{P}_{N-1}$ `-> "NAME" ->` $\texttt{P}_N$ `-> ...` $\texttt{P}_{N+M}$ `-> : TYPE`
- `NAME` is the name of the symbol
- `TYPE` indicates in what contexts this symbol can be used
- $\texttt{P}_1$ through $\texttt{P}_{N-1}$ are the parameters expected left of the symbol
- $\texttt{P}_N$ through $\texttt{P}_{N+M}$ are the parameters expected right of the symbol

```
Data Expr -> "|" -> Expr : Expr
```

Inference systems

Example: boolean disjunction

- `TRUE | TRUE` is a valid proposition
- `TRUE | FALSE` is a valid proposition
- `FALSE | TRUE` is a valid proposition
- `FALSE | FALSE` is a valid proposition
- `TRUE | (FALSE | FALSE)` is a valid proposition
- ...

Inference systems

Example: boolean disjunction

- Associativity is by default to the right
- `TRUE | TRUE | TRUE` is equivalent to `TRUE | (TRUE | TRUE)`
- We can modify associativity with `Associativity Left` in the symbol definition, for example with:
- `Data Expr -> "|" -> Expr : Expr Associativity Left`

Inference systems

Example: boolean disjunction

- Suppose we define boolean conjunction (the so-called *and* operator):
- `Data Expr -> "&" -> Expr : Expr`
- There is ambiguity with composite expressions such as `TRUE & FALSE | TRUE`:
  - `TRUE & (FALSE | TRUE)` or
  - `(TRUE & FALSE) | TRUE`?
- Quite dangerous: the two expressions mean different things!

Inference systems

Example: boolean disjunction

- We can specify a `Priority` of the symbols, for example with
- `Data Expr -> "|" -> Expr :  Expr Priority 10`
- `Data Expr -> "&" -> Expr :  Expr Priority 20`

Inference systems

Example: boolean disjunction

- So far we have defined `TRUE` and `FALSE` as `Expr`
- This is a bit imprecise, because an expression is usually composite
- We often call `TRUE` and `FALSE` *values*
- With the caveat that they can also be used wherever expressions are expected

Inference systems

Example: boolean disjunction

- We would like to `TRUE` with type `Value`
- But then also making it so that propositions of type `Value` can be used whenever proposition of type `Expr` are expected
- This is a form of *subtyping* (related to inheritance and polymorphism)

Polymorphism

```
Data "TRUE" : Value
Data "FALSE" : Value
```

Inference systems

Example: boolean disjunction

- Now `TRUE | FALSE` is not an acceptable composition
- `|` now expects `Expr`'s but we are giving it `Value`'s
- To fix this, we say that any `Value` is also an `Expr`

Polymorphism

```
Value is Expr
```

Inference systems

Example: boolean disjunction

- Any proposition of type `Value`...
- ...will be allowed whenever a proposition of type `Expr` is expected

Inference systems

Function symbols

- Function symbols are used to define transformations
- From one or more input propositions into one resulting proposition

Inference systems

Function symbols

- Functions are defined in two steps
- First we **declare** the function: name, parameters, and type
- Then we **define** the function: behaviour as a series of recursive definitions

Inference systems

Function declaration syntax

- `Func` $\texttt{P}_1$ `-> ...` $\texttt{P}_{N-1}$ `-> "NAME" ->` $\texttt{P}_N$ `-> ...` $\texttt{P}_{N+M}$ `-> : TYPE => RETURN`
- Function named `NAME`
- Parameters $\texttt{P}_1$ through $\texttt{P}_{N-1}$ to the left of the function
- Parameters $\texttt{P}_N$ through $\texttt{P}_{N+M}$
- Evaluation of the function will yield a result of type `RETURN` [a]

[a] The function with its parameters applied is a proposition of type `TYPE`

Inference systems

Function declaration syntax

- For example, we could define an `eval` function
- It takes as input a boolean expression of type `Expr`
- It returns as input a boolean value of type `Value`
- `Func "eval" -> Expr :  Expr => Value`

Inference systems

Function declaration syntax

- Valid uses of the function are:
- `eval TRUE`
- `eval FALSE`
- `eval (TRUE | TRUE)`
- ...

Inference systems

Rules and function evaluation

- We have only specified the acceptable *shape* of propositions and function calls
- We have not yet said anything about how computations happen

Inference systems

Syntax of rules

- Function evaluation is based on *rules*
- The rules of our language are syntactically quite simple
- The simplicity comes from the fact that a rule syntax is only made up of two operators:
  - the **horizontal bar** -------------, of varying length bigger than --
  - the **arrow** =>

Inference systems

Syntax of rules

- Both horizontal bar and implication arrow can be read out loud as “therefore”
- They both capture implication, as in “if A is true, then B is also true”
- The arrow specifies implication at a smaller scale
- The bar specifies implication at a larger scale

Structure of a rule

```
PREMISE 1
PREMISE 2
...
PREMISE N
-----------------
MAIN PROPOSITION
```

Inference systems

Syntax of rules

- Both premises and the main proposition are defined in terms of the arrow
- On the left of the arrow we have the input proposition
- On the right we have the output proposition
- `INPUT SYMBOLS => OUTPUT SYMBOLS`

Structure of a rule (more concrete)

```
I₁  =>  O₁
I₂  =>  O₂
:
I_N  =>  O_N
------------------
INPUT  =>  OUTPUT
```

Inference systems

Syntax of rules

- `INPUT` is the overall input of the rule
- It is always a function with parameters
- The rule works as follows:

  A certain proposition is found to match the input

  The premises are evaluated from top to bottom (first we evaluate $\texttt{I}_1$ to obtain $\texttt{O}_1$, then we move on to $\texttt{I}_2$ and $\texttt{O}_2$, etc.)

  `OUTPUT` is returned as the result

Inference systems

Syntax of rules

- The simplest rules feature no premises
- They directly specify how output is directly returned for a certain shape of input

Simplest rules example: evaluation of constants

```
-------------------
eval TRUE => TRUE
```

Inference systems

Syntax of rules

- The rule above looks for an input of shape `eval TRUE`
- When such an input is found, then we directly return `TRUE`

Another example

```
---------------------
eval FALSE => FALSE
```

Inference systems

Syntax of rules

- Sometimes a rule is more complex
- We cannot always determine the results directly from the input
- In this cases we need some further computation
- For this we use the rule premises

Evaluation of complex rules: boolean or

```
eval a => TRUE
-------------------
eval (a|b) => TRUE
```

Inference systems

Syntax of rules

- The rule above processes an `or`-expression
- Consider an `or` of two boolean expressions `a` and `b`
- We cannot directly determine if the whole `or` is true or false

Inference systems

Syntax of rules

- The first premise thus evaluates the first sub-expression with `eval a`
- We also say that we expect an output of `TRUE` from `eval a`
- We stop if `eval a` returns something else than `TRUE`
- Otherwise we return `TRUE` as the output of the rule itself

Inference systems

Syntax of rules

- What happens if `eval a` evaluates to `FALSE`
- The rule above cannot provide us with the answer we are looking for
- We simply look for another rule that can

Inference systems

Syntax of rules

- Complex behaviours
- Multiple rules
- `eval` is such an example

Evaluation of complex rules: boolean or

```
eval a => FALSE
eval b => y
-------------------
eval (a|b) => y
```

Inference systems

Syntax of rules

- The first premise evaluates the first sub-expression with `eval a`
- This time we expect `FALSE` as a result, thus this rule succeeds where the previous fails
- Should `eval a` return something else than `FALSE`, then we would stop
- If the output of the first premise is `FALSE`, then we proceed to the second premise
- Whatever result comes from `eval b` is returned as the output of the rule itself

Inference systems

Program evaluation

- An aspect that is still unclear is how rules are selected
- Rules specify a sort of library of possible evaluations
- We still miss an important ingredient: the initial input
- This input will (recursively) determine selection of rules
- The process either finds an output, or gets stuck with no answer

Inference systems

Program evaluation

- We now see the whole process in action
- We start from complete definitions for symbols and rules
- We then evaluate an input fully

Boolean expressions full example

```
Func "eval" -> Expr : Expr => Value
Data Expr -> "|" -> Expr : Expr

Data "TRUE" : Value
Data "FALSE" : Value

Value is Expr


-------------------
eval TRUE => TRUE

-------------------
eval FALSE => FALSE


eval a => TRUE
--------------------
eval (a|b) => TRUE

eval a => FALSE
eval b => y
--------------------
eval (a|b) => y
```

Inference systems

Program evaluation

- Suppose an initial input of `eval (FALSE | TRUE)`
- This becomes our *current proposition*
- The first step is to find **matching rules**, those rules that "could be used" to evaluate the current proposition

Inference systems

Program evaluation

- Matching only looks at the input of rules
- It looks for rules which input can be mapped to our current proposition
- The candidate rule inputs are, in order of declaration:
  - `eval TRUE`
  - `eval FALSE`
  - `eval (a|b)`
  - `eval (a|b)`

Inference systems

Program evaluation

- We set the current proposition and the input of each rule side-by-side
- When we find a rule input that could be transformed into the current proposition, we are done
- *Could be transformed* through an assignment of the variables, and no more

First input vs current proposition

We try the first rule:

```
eval        TRUE
eval  (FALSE | TRUE)
```

First input vs current proposition

We try the first rule:

```
eval      TRUE
eval (FALSE | TRUE)
```

`eval` matches, but the parameters are clearly different and unrelated

The first rule cannot be used in this case.

Second input vs current proposition

We try the second rule:

```
eval      FALSE
eval (FALSE | TRUE)
```

Second input vs current proposition

We try the second rule:

```
eval       FALSE
eval  (FALSE | TRUE)
```

`eval` matches, but the parameters are clearly different and unrelated

The second rule cannot be used in this case.

Third input vs current proposition

We try the second rule:

```
eval (  a    |  b  )
eval (FALSE | TRUE)
```

Third input vs current proposition

We try the second rule:

```
eval (  a    |  b  )
eval (FALSE | TRUE)
```

We have a match!

The current proposition matches the third rule, **provided that** `a=FALSE` **and** `b=TRUE`

Inference systems

Program evaluation

- a=FALSE and b=TRUE is called a **binding** of the rule variables
- It binds the variables of the rule input, which so far were unspecified, to parts of the current proposition

We represent this process graphically by putting the current proposition underneath the rule:

```
eval a => TRUE
----------------------------
eval (  a   |  b  ) => TRUE
eval (FALSE | TRUE)
```

We know that a=`FALSE` and b=`TRUE`, so we replace the variables with their values:

```
eval FALSE => TRUE
------------------------------
eval (FALSE | TRUE) => TRUE
```

Inference systems

Program evaluation

- Now we need to make sure that all premises are valid
- We start with the first premise, `eval FALSE => TRUE`
- We take its input `eval FALSE`, and consider it to be the new current proposition

We can represent this graphically by re-writing the current proposition above the premise we are evaluating:

```
eval FALSE
eval FALSE => TRUE
-----------------------------
eval (FALSE | TRUE) => TRUE
```

Inference systems

With the new *current proposition*

- We repeat the matching process
- When the matching process is completed, we evaluate the matched rule recursively
- When evaluation of the rule is completed, we match the resulting output with the expected output
- After this last match is completed, we proceed with the evaluation of the original rule

We will represent this graphically by writing the rule we are considering above the current proposition to see if it matches.

We start with the first rule:

```
------------------
eval TRUE  => TRUE
eval FALSE
eval FALSE => TRUE
----------------------------
eval (FALSE | TRUE) => TRUE
```

We will represent this graphically by writing the rule we are considering above the current proposition to see if it matches.

We start with the first rule:

```
------------------
eval TRUE  => TRUE
eval FALSE
eval FALSE => TRUE
-----------------------------
eval (FALSE | TRUE) => TRUE
```

Clearly no match

We move on to the second rule:

```
------------------
eval FALSE => FALSE
eval FALSE
eval FALSE => TRUE
----------------------------
eval (FALSE | TRUE) => TRUE
```

We move on to the second rule:

```
-------------------
eval FALSE => FALSE
eval FALSE
eval FALSE => TRUE
-----------------------------
eval (FALSE | TRUE) => TRUE
```

We have a clear match!

The matched rule immediately tells us that the result was (`FALSE`), so we need no further processing above.

We take the result of FALSE and write it next to the current proposition.

We can also delete the uppermost rule, because its evaluation is completed and we do not need it any more.

We also try to match the actual output of the premise to the expected output:

```
eval FALSE => FALSE
eval FALSE => TRUE
----------------------------
eval (FALSE | TRUE) => TRUE
```

We take the result of `FALSE` and write it next to the current proposition.

We can also delete the uppermost rule, because its evaluation is completed and we do not need it any more.

We also try to match the actual output of the premise to the expected output:

```
eval FALSE => FALSE
eval FALSE => TRUE
----------------------------
eval (FALSE | TRUE) => TRUE
```

No match!

We abort evaluation of the whole rule, and go back almost to the beginning.

Inference systems

With the new *current proposition*

- We go back to the original current proposition of `eval (FALSE | TRUE)`
- We do not go entirely to the beginning, because we still know that:
  - the first rule does not match
  - the second rule does not match
  - the third rule matches, but gets stuck at the first premise
- so we proceed with the fourth rule

We write the fourth rule above the current preposition:

```
eval  a  =>  FALSE
eval  b  =>  y
-----------------------------
eval  (   a    |   b   )  =>  TRUE
eval  (FALSE  |  TRUE)
```

We write the fourth rule above the current preposition:

```
eval a => FALSE
eval b => y
-----------------------------
eval (  a   |  b  ) => TRUE
eval (FALSE | TRUE)
```

We have a match, with the same binding as for the third rule: a=FALSE and b=TRUE.

We replace a and b in the rule with the bound values:

```
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
```

We evaluate the first premise, so we duplicate its input to signal that we now have a new current proposition:

```
eval FALSE
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
```

We try to match the first rule to the (new) current proposition:

```
------------------
eval TRUE => TRUE
eval FALSE
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
```

We try to match the first rule to the (new) current proposition:

```
------------------
eval TRUE => TRUE
eval FALSE
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
```

No success! Moving on to the next rule.

We try to match the second rule to the (new) current proposition:

```
------------------
eval FALSE => FALSE
eval FALSE
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
```

We try to match the second rule to the (new) current proposition:

```
------------------
eval FALSE => FALSE
eval FALSE
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
```

Success! The current proposition results in `FALSE`, so we delete the upper rule (it is done evaluating and we do not need it anymore) and propagate its result down.

We now try to match the actual result of the premise with the actual result of the premise:

```
eval FALSE => FALSE
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
```

We now try to match the actual result of the premise with the actual result of the premise:

```
eval FALSE => FALSE
eval FALSE => FALSE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
```

Both are `FALSE`, so matching is trivial.

We delete the upper lines because they are done evaluating, and have no further use:

```
eval TRUE => y
------------------------------
eval (FALSE | TRUE) => y
```

We must now evaluate the last premise, `eval TRUE => y`.

The new *current proposition* is `eval TRUE`:

```
-------------------
eval TRUE
eval TRUE => y
------------------------------
eval (FALSE | TRUE) => y
```

We try matching the first rule:

```
------------------
eval TRUE => TRUE
eval TRUE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
```

We try matching the first rule:

```
------------------
eval TRUE => TRUE
eval TRUE
eval TRUE => y
----------------------------
eval (FALSE | TRUE) => y
```

The first rule succesfully matches and immediately returns TRUE.

We propagate the result down next to the premise input:

```
eval TRUE => TRUE
eval TRUE =>  y
----------------------------
eval (FALSE | TRUE) => y
```

Now we propagate the result (TRUE) down to the premise (y) with another matching:

```
eval TRUE => TRUE
eval TRUE => TRUE
----------------------------
eval (FALSE | TRUE) => TRUE
```

Now we propagate the result (TRUE) down to the premise (y) with another matching:

```
eval TRUE => TRUE
eval TRUE => TRUE
-----------------------------
eval (FALSE | TRUE) => TRUE
```

This matching also succeeds, with binding y=TRUE.

Now that we have a value for y, we can replace all its occurrences with its value TRUE.

We delete the premises above, since they have no further use, and voilá:

```
------------------------------
eval (FALSE | TRUE) => TRUE
```

We delete the premises above, since they have no further use, and voilá:

```
------------------------------
eval (FALSE | TRUE) => TRUE
```

We now know the result of the evaluation of the initial proposition. This result is also what we expected, to our satisfaction.

Inference systems

Demo-time!

- Visual Studio 2015
- Standalone GUI

Assignment 1

Assignment 1

In small groups

- One rule per post-it
- Start with a proposition
- Try to put the rule above the proposition and see if it matches
- Success -¿ go forward with rule propositions
- Failure -¿ try another rule
- Do it with boolean expressions

Assignment 2

Assignment 2

In small groups

- Add the XOR operator to the boolean expressions sample
- Define and add a custom operator to the boolean expressions sample

Conclusions

Conclusions

Wrapping-up

- We have introduced logical reasoning
- We have seen a first intuition of logic, symbols, and computation
- We have seen a first example of using logic to define boolean expressions and associated computations