

**0.0.1 Example.** Russell's Paradox

We represent sets either by *listing*,

- $\{0\}$
- $\{\$, \#, 3, 42\}$
- $\{0, 1, 2, 3, 4, \dots\}$

or by “*defining property*”: *The set of all objects  $x$  that make  $P(x)$  true*, in symbols

$$S = \{x : P(x)\} \tag{1}$$

As we know from high-school, (1) says the *same thing*, or as we say, *is equivalent to*

$$x \in S \equiv P(x) \tag{2}$$

BTW, saying “all  $x$  (such) that ...” means “all values of  $x$  (such) that ...”

*Moreover, writing “ $P(x)$ ” is the same as writing “ $P(x)$  is true”.*

---

Cantor believed (as did the philosopher Frege) that, for any property  $P(x)$ , (1) defines a set.

Russell begged to differ, so he said: “Oh, yeah? How about”

$$R = \{x : x \notin x\}$$

where the property here is “ $x \notin x$ ”

Now, by (2) we have

$$x \in R \equiv x \notin x$$

If  $R$  is a set, then we can **plug it in the set variable  $x$**  above to obtain

$$R \in R \equiv R \notin R$$

How do we avoid this *contradiction*?


By *admitting* that  $R$  is *NOT* a set!



□

---

So Cantor was sloppy about what a set is and how sets get formed.

Formal logic was invented by Russell and Whitehead, and Hilbert to salvage Mathematics from “antinomies” and “paradoxes”, both words derived from Greek, and meaning **contradictions**.

In the text, a passage enclosed between two “” signs is IMPORTANT.

A passage enclosed between two double-signs, “” is to be *omitted* at first reading.

## Connection with Programming

- (1) In programming we use *syntactic rules* to write a *program* in order to solve some problem computationally.
  
- (2) In logic you use the syntactic rules to write a *proof* that establishes a theorem.

Kinds of logic reasoning that we will thoroughly examine and use.

1. Equational logic —also known as *calculational logic*.

Introduced by [DS90] and simplified by [GS94] and later by [Tou08] to make it accessible to undergraduates.

2. Hilbert-style logic. This is the logic which most people use to write their mathematical arguments in publications, lectures, etc.

---

Logic is meant to certify mathematical truths syntactically.

**Logic is normally learnt by**

- A *LOT* of practice.
- By presenting it *gradually*.
  1. **First learning the *Propositional Logic* (also known as *Boolean Logic*).**

Here one learns how logical truths *combine* using connectives familiar from programming like OR, AND, and NOT.

Boolean logic is *not expressive enough* to formulate statements about mathematical objects. Naturally, if you cannot ask it —a question about such objects— then you cannot answer it either.

2. **Next learning *Predicate Logic* (also known as *First-Order Logic*).**

This is the full logic for the mathematician and computer scientist as it lets you formulate and explore statements that involve *mathematical objects* like numbers, strings and trees, and many others.

**Decision Problem of Logic** (*Entscheidungsproblem* of Hilbert's): It asks: **Is this formula a theorem of logic?**



Here we are ahead of ourselves: What is a “**formula**”? What is a “**theorem**”?

I will tell you soon!

1. **Boolean Logic:** Its Decision Problem, because of *Post's theorem* that we will learn in this course, *does* have an *algorithmic solution*, for example, via *truth tables*.

There is a catch: The solution is in general useless because the algorithm takes tons of time to give an answer.

*I am saying that at the present state of knowledge of algorithms, the truth table method is unpractically slow. To get an answer from a  $n \times n$  table it takes  $2^n$  steps.*

2. **Predicate Logic:** Things get desperate here: We have a totally negative answer to the Decision Problem. *There is no algorithm* at all that will solve it! This result is due to Church ([Chu36])



So it *makes sense* to find ways to certify truth, which rely on human ingenuity and sound methodology rather than on some machine and a computer program, and do so both for

- Boolean logic where the decision problem has an *unfeasible* algorithm that solves it,  
and
- Predicate logic where the decision problem has *no algorithm to be discovered* —ever.

This we will learn in this course: How to certify truth by syntactic means, through practice and sound methodology.



### 0.0.1 Lecture #2 (Sep. 11)

## A look back at strings

### 0.0.2 Definition. (Strings; also called Expressions)

1. What is a *string* over some *alphabet* of symbols?

It is an *ordered* finite sequence of symbols from the alphabet —with no gaps between symbols.

**0.0.3 Example.** If the alphabet is  $\{a, b\}$  then here are a few strings:

- (a)  $a$
- (b)  $aaabb$
- (c)  $baaaa$
- (d)  $bbbbbb$

□

What do we mean by “ordered”? We mean that order matters! For example,  $aaabb$  and  $baaaa$  are different strings. We indicate this by writing  $aaabb \neq baaaa$ .



Two strings are equal iff <sup>†</sup> they have the same length  $n$  and at *each* position —from 1 to  $n$ — both strings have the same symbol. So,  $aba = aba$ , but  $aa \neq a$  and  $aba \neq baa$ .




---

<sup>†</sup>If and only if.



## A Bad Alphabet

Consider the alphabet  $B = \{a, aa\}$ .

This is bad. WHY?

Because if we write the string **aaa** over this alphabet we do not know what we mean by just **looking at the string!**

Do we mean 3  $a$  like

$a \ a \ a$

Or do we mean

$a \ aa$

Or perhaps

$aa \ a$

We say that alphabet  $B$  leads to *ambiguity*.

*Since we use NO separators between symbols in denoting strings we MUST ALWAYS choose alphabets with single-symbol items.*

2. Names of strings:  $A, A'', A_5, B, C, S, T$ .

*What for?* CONVENIENCE AND EASE OF EXPRESSION.

Thus  $A = bba$  gives the string  $bba$  the name  $A$ .

*Names vs IS*: Practicing mathematicians and computer scientists take a sloppy attitude in using “IS”. In the case of “let  $A$  be a string” they mean “let  $A$  name a string”.

Same as in “let  $x$  be a rational number”. Well  $x$  is not a number at all! It is a letter! We mean “let  $x$  STAND for, or NAME, a rational number”

3. Operations on strings: Concatenation. From a string  $aab$  and  $baa$ , concatenation in the order given is  $aabbaa$ .

We say that if  $A$  is a string (meaning names) and  $B$  is another. Their concatenation  $AB$  is not a concatenation of names but a concatenation of contents. If  $A = aaaa$  and  $B = 101$  then  $AB = aaaa101$ .

Incidentally,

$$BA = 101aaaa \neq aaaa101 = AB$$

Thus in *general* concatenation is *not commutative* as we say.

Why “in general”?

Well, if  $X = aa$  and  $Y = a$  then  $XY = aaa = YX$ .

*Special cases* where concatenation commutes exist!

4. Associativity of concatenation. It is expressed as  $(AB)C = A(BC)$  where bracketing here denote invisible symbols (*not part of any* string!) that simply indicate **the order in which we GROUP, from left to right**.

At the left of the “=” we first concatenate  $A$  and  $B$  and then glue  $C$  at the *right* end.

To the right of “=” we first glue  $B$  and  $C$  and then glue  $A$  to the *left* of the result.

In either case we did not change the relative positions of  $A$ ,  $B$  and  $C$ .

The property is self-evident.

- 
5. Empty string. A string with *no symbols*, hence with *length 0*. Denoted by  $\lambda$ .



How is  $\lambda$  different than  $\emptyset$  the empty *set*?

Well one is of *string type* and the other is of *set type*. So? The former is an ORDERED empty set, the latter is an UNORDERED empty set that moreover is *oblivious to repetitions*.

I mean,  $aaa \neq a$  but  $\{a, a, a\} = \{a\}$ .



6. Clearly, for any string  $A$  we have  $A\lambda = \lambda A$  as concatenation adds nothing to either end.

7. Substrings. A string  $A$  is a *substring* of  $B$  iff  $A$  appears as is as a *part* of  $B$ .

So if  $A = aa$  and  $B = aba$  then  $A$  is NOT a substring of  $B$ .

Its members both appear in  $B$  (the two  $a$ ) but are *not* together as they are in  $A$ .  $A$  *does not appear "as is"*.

**Can we get rid of all this bla-bla with a proper definition? Sure:**

$A$  is a substring of  $B$  iff for some strings (named)  $U$  and  $V$  we have  $B = UAV$ .



We also say  $A$  is *part* of  $B$ .



8. Prefix and suffix.  $A$  is a *prefix* of  $B$  if for some string  $V$ ,  $B = AV$ .

So  $A$  is part of  $B$  up in front!

$A$  is a *suffix* of  $B$  if for some string  $U$ ,  $B = UA$ .

□

**Example:**  $\lambda$  is a prefix and a suffix, indeed a part, of *any* string  $B$ . Here are the “proofs” of the two cases I enumerated:

- $B = \lambda B$
- $B = B\lambda$

WHAT ABOUT THE THIRD CASE?

# Lecture #3 (Sept. 16) Formulas or well-formed-formulas (wff)

## The Syntax of logic. Boolean Logic at first!

### 0.0.4 Definition. (Alphabet of Symbols)

**A1.** Names for *variables*, which we call “propositional” or “Boolean” variables.

These are  $p, q, r$ , with or without primes or subscripts (indices) (e.g.,  $p, q, r, p', q_{13}, r'''_{51}$  are all names for Boolean variables).

**A2.** Two *symbols* denote the Boolean *constants*,  $\top$  and  $\perp$ . We pronounce them “top” and “bot” respectively.

What are  $\top$  and  $\perp$  good for? We will soon see!

**A3.** (Round) brackets, i.e., “(” and “)” (employed without the quotes, of course).

**A4.** Boolean “*connectives*” that I will usually call “*glue*”.

We use glue to put a formula together much like we do so when we build model cars or airplanes.

The symbols for Boolean connectives are

$$\neg \wedge \vee \rightarrow \equiv \tag{1}$$

and are read from left to right as “negation, conjunction, disjunction, implication, equivalence”.  $\square$



We stick to the above symbols for glue (no pun!) in this course! *Just like in programming.* You *cannot use* any symbols you **please** or **like**. You use **THE** symbols of the programming language as **GIVEN**. Same holds for logic!






**0.0.5 Definition. (Formula Construction (process))** A *formula construction* (in the text “*formula calculation*”) is any finite (ordered) sequence of strings the alphabet of Boolean logic  $\mathcal{V}$  that obeys the following three specifications:

- C1. At any step we may write precisely one symbol from categories **A1.** or **A2.** above (0.0.4).
- C2. At any step we may write precisely *one string* of the form  $(\neg A)$ , as long as we have written the string (*named!*)  $A$  already at a previous step.

So, “ $(\neg A)$ ” is a string that has  $(\neg$  as a prefix, then it has a part we named  $A$ , and then it has  $)$  as a suffix.

 I must stress that the letter  $A$  **names** the string that we write down. Just as in a *program*: When you issue the command “**print**  $X$ ” you mean to **print what the  $X$  contains as value** —**what it names**. You do *not* mean to print the **letter** “ $X$ ”!



- C3. At any step we may write precisely *one of the strings*  $(A \wedge B)$ ,  $(A \vee B)$ ,  $(A \rightarrow B)$ ,  $(A \equiv B)$ , as long as we have already written *each* of the strings  $A$  and  $B$  earlier.

 We do not care which we wrote first,  $A$  or  $B$ .



□

**0.0.6 Definition. (Boolean or Propositional formulas (wff))** Any string  $A$  over the alphabet  $\mathcal{V}$  (**A1.**–**A4.**) is called a *Boolean formula* or *propositional formula*—in short **wff**—iff  $A$  is a string that **appears** in some formula construction.  $\square$

**0.0.7 Example.** First off, the above says more than it pretends to:

For example, it says that *every* string that appears in a formula construction is a wff. **The definition also says,**

*“do you want to know if  $A$  is a wff? Just make sure you can build a formula construction where  $A$  appears.”*

We normally write formula constructions **vertically**. Below I use numbering and annotation (in “ $\langle \cdot \rangle$ ” brackets) to explain each step.

•

- (1)  $\perp$        $\langle C1 \rangle$
- (2)  $p$          $\langle C1 \rangle$
- (3)  $(\neg\perp)$     $\langle (1) + (C2) \rangle$
- (4)  $\perp$          $\langle C1 \rangle$
- (5)  $\top$          $\langle C1 \rangle$

Note that we can have redundancy and repetitions. Ostensibly the only nontrivial info in the above is that  $(\neg\perp)$  is a formula. But it also establishes that  $\perp$  and  $\top$  and  $p$  are formulas.

•

- (1)  $\perp$          $\langle C1 \rangle$
- (2)  $p$          $\langle C1 \rangle$
- (3)  $(\neg\top)$     $\langle \text{oops!} \rangle$
- (4)  $\perp$          $\langle C1 \rangle$
- (5)  $\top$          $\langle C1 \rangle$

The above is wrong at step (3). I have not written  $\top$  in the construction *before* I attempted to use it!

□



# Bibliography

- [Chu36] Alonzo Church, *A note on the Entscheidungsproblem*, J. Symbolic Logic **1** (1936), 40–41, 101–102.
- [DS90] Edsger W. Dijkstra and Carel S. Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag, New York, 1990.
- [GS94] David Gries and Fred B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, New York, 1994.
- [Tou08] G. Tourlakis, *Mathematical Logic*, John Wiley & Sons, Hoboken, NJ, 2008.