Metalogic

Fall Semester 2024

András Máté

Eötvös Loránd University Budapest
Institute of Philosophy, Department of Logic
mate.andras53@gmail.com

13.09.2024

General

- Source (= textbook):
  Ruzsa, I., *Introduction to Metalogic*. Budapest: 2on Publishers, 1997.

General

- Source (= textbook):
  Ruzsa, I., *Introduction to Metalogic*. Budapest: 2on Publishers, 1997.
- Necessary preliminary knowledge:
  In principle: nothing.
  In practice: the language of first-order logic/set theory.

General

- Source (= textbook):
  Ruzsa, I., *Introduction to Metalogic*. Budapest: 2on Publishers, 1997.
- Necessary preliminary knowledge:
  In principle: nothing.
  In practice: the language of first-order logic/set theory.
- Method: lecture + solving problems.

General

- Source (= textbook):
  Ruzsa, I., *Introduction to Metalogic*. Budapest: 2on Publishers, 1997.
- Necessary preliminary knowledge:
  In principle: nothing.
  In practice: the language of first-order logic/set theory.
- Method: lecture + solving problems.
- Evaluation: solving problems (during the classes or in the exam period).

General

- Source (= textbook):
  Ruzsa, I., *Introduction to Metalogic*. Budapest: 2on Publishers, 1997.
- Necessary preliminary knowledge:
  In principle: nothing.
  In practice: the language of first-order logic/set theory.
- Method: lecture + solving problems.
- Evaluation: solving problems (during the classes or in the exam period).
- Webpage:
  http://phil.elte.hu/mate/metalogic/metalogic.html.
  Presentations (pdf-s) will be published after the classes.

What is this?

What is this?

Metalogic: Logical theory of logical theories and theories formalized in logic.

What is this?

Metalogic: Logical theory of logical theories and theories formalized in logic.
Metalogical theorems: deductive completeness, algorithmic undecidability of first order logic; negation-incompleteness of first-order Peano arithmetic, etc.

What is this?

Metalogic: Logical theory of logical theories and theories formalized in logic.
Metalogical theorems: deductive completeness, algorithmic undecidability of first order logic; negation-incompleteness of first-order Peano arithmetic, etc.
Aim: to build a theory such that

What is this?

Metalogic: Logical theory of logical theories and theories formalized in logic.
Metalogical theorems: deductive completeness, algorithmic undecidability of first order logic; negation-incompleteness of first-order Peano arithmetic, etc.
Aim: to build a theory such that

- it proves such theorems in an abstract form (i. e., about a large family of theories) and in an unique framework.

What is this?

Metalogic: Logical theory of logical theories and theories formalized in logic.
Metalogical theorems: deductive completeness, algorithmic undecidability of first order logic; negation-incompleteness of first-order Peano arithmetic, etc.
Aim: to build a theory such that

- it proves such theorems in an abstract form (i. e., about a large family of theories) and in an unique framework.
- it can serve as a foundation of other logical theories: it doesn't use them and it gives a clear account about its presuppositions.

What is this?

Metalogic: Logical theory of logical theories and theories formalized in logic.
Metalogical theorems: deductive completeness, algorithmic undecidability of first order logic; negation-incompleteness of first-order Peano arithmetic, etc.
Aim: to build a theory such that

- it proves such theorems in an abstract form (i. e., about a large family of theories) and in an unique framework.
- it can serve as a foundation of other logical theories: it doesn't use them and it gives a clear account about its presuppositions.

Circularity- ('hen and egg'-)problem in foundations: the relation between syntax (proof theory) and semantics (set theory).

What is this?

Metalogic: Logical theory of logical theories and theories formalized in logic.
Metalogical theorems: deductive completeness, algorithmic undecidability of first order logic; negation-incompleteness of first-order Peano arithmetic, etc.
Aim: to build a theory such that

- it proves such theorems in an abstract form (i. e., about a large family of theories) and in an unique framework.
- it can serve as a foundation of other logical theories: it doesn't use them and it gives a clear account about its presuppositions.

Circularity- ('hen and egg'-)problem in foundations: the relation between syntax (proof theory) and semantics (set theory).
Our theory: that of *canonical calculi* (including Markov algorithms).

A metalanguage for formal languages

Metalanguage:

A metalanguage for formal languages

Metalanguage:

- A fragment of the communication language

A metalanguage for formal languages

Metalanguage:

- A fragment of the communication language
- extended by some formal tools

A metalanguage for formal languages

Metalanguage:

- A fragment of the communication language
- extended by some formal tools

that suffices to formulate our theory.

A metalanguage for formal languages

Metalanguage:

- A fragment of the communication language
- extended by some formal tools

that suffices to formulate our theory.

Expressions of our metalanguage:

A metalanguage for formal languages

Metalanguage:

- A fragment of the communication language
- extended by some formal tools

that suffices to formulate our theory.

Expressions of our metalanguage:

- sentences;

A metalanguage for formal languages

Metalanguage:

- A fragment of the communication language
- extended by some formal tools

that suffices to formulate our theory.

Expressions of our metalanguage:

- sentences;
- names;

A metalanguage for formal languages

Metalanguage:

- A fragment of the communication language
- extended by some formal tools

that suffices to formulate our theory.

Expressions of our metalanguage:

- sentences;
- names;
- functors.

A metalanguage for formal languages

Metalanguage:

- A fragment of the communication language
- extended by some formal tools

that suffices to formulate our theory.

Expressions of our metalanguage:

- sentences;
- names;
- functors.

Functors are expressions containing empty places (argument places) that can be filled in by some expressions of definite type (arguments). By filling in the empty places we gain a new expression of some definite type.

A metalanguage for formal languages

Metalanguage:

- A fragment of the communication language
- extended by some formal tools

that suffices to formulate our theory.
Expressions of our metalanguage:

- sentences;
- names;
- functors.

Functors are expressions containing empty places (argument places) that can be filled in by some expressions of definite type (arguments). By filling in the empty places we gain a new expression of some definite type.
I.e., it is assumed that there is one and only type assigned to each empty place of a functor.

A metalanguage for formal languages

Metalanguage:

- A fragment of the communication language
- extended by some formal tools

that suffices to formulate our theory.

Expressions of our metalanguage:

- sentences;
- names;
- functors.

Functors are expressions containing empty places (argument places) that can be filled in by some expressions of definite type (arguments). By filling in the empty places we gain a new expression of some definite type.

I.e., it is assumed that there is one and only type assigned to each empty place of a functor.

Or: a functor is an *automaton* taking expressions as inputs and producing another expression as the output.

Types (groups) of functors

Types (groups) of functors

A functor is called monadic, dyadic, . . . , n-adic according to the number of the empty places.

Types (groups) of functors

A functor is called monadic, dyadic, ..., n-adic according to the number of the empty places.
A functor is homogeneous if the same type is assigned to each empty place.

Types (groups) of functors

A functor is called monadic, dyadic, . . . , n-adic according to the number of the empty places.
A functor is homogeneous if the same type is assigned to each empty place.
Types (families) of homogeneous functors:

Types (groups) of functors

A functor is called monadic, dyadic, . . . , n-adic according to the number of the empty places.
A functor is homogeneous if the same type is assigned to each empty place.
Types (families) of homogeneous functors:

- sentential functors take sentences as arguments and give a sentence;

Types (groups) of functors

A functor is called monadic, dyadic, . . . , n-adic according to the number of the empty places.
A functor is homogeneous if the same type is assigned to each empty place.
Types (families) of homogeneous functors:

- sentential functors take sentences as arguments and give a sentence;
- name functors take names and give a name;

Types (groups) of functors

A functor is called monadic, dyadic, . . . , n-adic according to the number of the empty places.
A functor is homogeneous if the same type is assigned to each empty place.
Types (families) of homogeneous functors:

- sentential functors take sentences as arguments and give a sentence;
- name functors take names and give a name;
- predicates take names and give a sentence.

Types (groups) of functors

A functor is called monadic, dyadic, . . . , n-adic according to the number of the empty places.
A functor is homogeneous if the same type is assigned to each empty place.
Types (families) of homogeneous functors:

- sentential functors take sentences as arguments and give a sentence;
- name functors take names and give a name;
- predicates take names and give a sentence.

A special dyadic name functor: concatenation ($^\frown$). If a and b are two strings, then $a^\frown b$ is the string beginning with a and continued by b.

Types (groups) of functors

A functor is called monadic, dyadic, ..., n-adic according to the number of the empty places.
A functor is homogeneous if the same type is assigned to each empty place.
Types (families) of homogeneous functors:

- sentential functors take sentences as arguments and give a sentence;
- name functors take names and give a name;
- predicates take names and give a sentence.

A special dyadic name functor: concatenation ($^\frown$). If a and b are two strings, then $a^\frown b$ is the string beginning with a and continued by b.
A special dyadic predicate: identity. If a and b are strings, then $a = b$ is the sentence saying that a and b are the same string.

Logical tools I.: quantifiers and variables

Logical tools I.: quantifiers and variables

To express sentences containing expressions like ‘every’, ‘each’, some’, etc. we use quantifiers and variables.

.

To express sentences containing expressions like ‘every’, ‘each’, some’, etc. we use quantifiers and variables.

Sentences of the form ⌜Every A is a B⌝ should be rewritten as ⌜For every x, if x is an A, then x is a B⌝

.

Logical tools I.: quantifiers and variables

To express sentences containing expressions like ‘every’, ‘each’, some’, etc. we use quantifiers and variables.

Sentences of the form ⌜Every A is a B⌝ should be rewritten as ⌜For every x, if x is an A, then x is a B⌝

Abbreviation: ⌜$\bigwedge x$(if x is an A, then x is a B)⌝

.

Logical tools I.: quantifiers and variables

To express sentences containing expressions like ‘every’, ‘each’, some’, etc. we use quantifiers and variables.

Sentences of the form $\ulcorner$Every A is a $B\urcorner$ should be rewritten as $\ulcorner$For every x, if x is an A, then x is a $B\urcorner$

Abbreviation: $\ulcorner \bigwedge x($if x is an A, then x is a $B)\urcorner$

$\ulcorner$Some A is a $B\urcorner$: $\ulcorner \bigvee x(x$ is an A and x is a $B)\urcorner$.

To express sentences containing expressions like 'every', 'each', some', etc. we use quantifiers and variables.

Sentences of the form ⌜Every A is a B⌝ should be rewritten as ⌜For every x, if x is an A, then x is a B⌝

Abbreviation: ⌜$\bigwedge x$(if x is an A, then x is a B)⌝

⌜Some A is a B⌝: ⌜$\bigvee x$(x is an A and x is a B)⌝.

We have *an unlimited number of* variables.

Logical tools I.: quantifiers and variables

To express sentences containing expressions like ‘every’, ‘each’, some’, etc. we use quantifiers and variables.

Sentences of the form $\ulcorner$Every A is a $B\urcorner$ should be rewritten as $\ulcorner$For every x, if x is an A, then x is a $B\urcorner$

Abbreviation: $\ulcorner\bigwedge x$(if x is an A, then x is a B)$\urcorner$

$\ulcorner$Some A is a $B\urcorner$: $\ulcorner\bigvee x$(x is an A and x is a B)$\urcorner$.

We have *an unlimited number of* variables.

They can occur in sentences at any place where names can occur.

Logical tools I.: quantifiers and variables

To express sentences containing expressions like ‘every’, ‘each’, some’, etc. we use quantifiers and variables.

Sentences of the form ⌜Every A is a B⌝ should be rewritten as ⌜For every x, if x is an A, then x is a B⌝

Abbreviation: ⌜$\bigwedge x$(if x is an A, then x is a B)⌝

⌜Some A is a B⌝: ⌜$\bigvee x$(x is an A and x is a B)⌝.

We have *an unlimited number of* variables.

They can occur in sentences at any place where names can occur.

Plus: they can occur in quantifying expressions (QE-s) consisting of a quantifier ($\bigvee$ or $\bigwedge$) and a variable.

QE-s in general

QE-s in general

QE-s are monadic sentential functors. The argument of a QE is called its scope. The variable of a QE makes the occurrences in its scope bounded. Other occurrences are free.

QE-s in general

QE-s are monadic sentential functors. The argument of a QE is called its scope. The variable of a QE makes the occurrences in its scope bounded. Other occurrences are free.

Sentences containing no free variable occurrences are closed, other sentences are open. Names can be closed resp. open, too.

QE-s in general

QE-s are monadic sentential functors. The argument of a QE is called its scope. The variable of a QE makes the occurrences in its scope bounded. Other occurrences are free.

Sentences containing no free variable occurrences are closed, other sentences are open. Names can be closed resp. open, too.

$\bigwedge xA$ is true iff any *substitution* of the quotation name of a string for x into A gives a true sentence.

QE-s in general

QE-s are monadic sentential functors. The argument of a QE is called its scope. The variable of a QE makes the occurrences in its scope bounded. Other occurrences are free.

Sentences containing no free variable occurrences are closed, other sentences are open. Names can be closed resp. open, too.

$\bigwedge xA$ is true iff any *substitution* of the quotation name of a string for x into A gives a true sentence.

$\bigvee xA$ is true iff at least one substitution of the quotation name of a string for x into A gives a true sentence.

QE-s in general

QE-s are monadic sentential functors. The argument of a QE is called its scope. The variable of a QE makes the occurrences in its scope bounded. Other occurrences are free.

Sentences containing no free variable occurrences are closed, other sentences are open. Names can be closed resp. open, too.

$\bigwedge xA$ is true iff any *substitution* of the quotation name of a string for x into A gives a true sentence.

$\bigvee xA$ is true iff at least one substitution of the quotation name of a string for x into A gives a true sentence.

The *intended universe* of this metalanguage is the class of finite strings of letters of some finite alphabet. Quantification is defined by substitution and by this, we are not committed to the existence of some set-theoretic universe built on this class.

We use the following abbreviations for some sentential functors of the metalanguage (A and B are sentences of the metalanguage):

Logical tools 2.: Logical sentential functors

We use the following abbreviations for some sentential functors of the metalanguage (A and B are sentences of the metalanguage):

- $\neg A$ for $\ulcorner$It is not true that $A\urcorner$;

We use the following abbreviations for some sentential functors of the metalanguage (A and B are sentences of the metalanguage):

- $\neg A$ for ⌜It is not true that A⌝;
- $A \wedge B$ for ⌜A and B⌝;

We use the following abbreviations for some sentential functors of the metalanguage (A and B are sentences of the metalanguage):

- $\neg A$ for ⌜It is not true that A⌝;
- $A \wedge B$ for ⌜A and B⌝;
- $A \vee B$ for ⌜A or B⌝;

Logical tools 2.: Logical sentential functors

We use the following abbreviations for some sentential functors of the metalanguage (A and B are sentences of the metalanguage):

- $\neg A$ for $\ulcorner$It is not true that $A\urcorner$;
- $A \wedge B$ for $\ulcorner A$ and $B\urcorner$;
- $A \vee B$ for $\ulcorner A$ or $B\urcorner$;
- $A \Rightarrow B$ for $\ulcorner$ If A, then $B\urcorner$;

Logical tools 2.: Logical sentential functors

We use the following abbreviations for some sentential functors of the metalanguage (A and B are sentences of the metalanguage):

- $\neg A$ for ⌜It is not true that A⌝;
- $A \wedge B$ for ⌜A and B⌝;
- $A \vee B$ for ⌜A or B⌝;
- $A \Rightarrow B$ for ⌜ If A, then B⌝;
- $A \Leftrightarrow B$ for ⌜A if and only if B⌝.

We use the following abbreviations for some sentential functors of the metalanguage (A and B are sentences of the metalanguage):

- $\neg A$ for ⌜It is not true that A⌝;
- $A \wedge B$ for ⌜A and B⌝;
- $A \vee B$ for ⌜A or B⌝;
- $A \Rightarrow B$ for ⌜ If A, then B⌝;
- $A \Leftrightarrow B$ for ⌜A if and only if B⌝.

If you studied classical propositional logic, use the truth conditions learned there for such sentences.

Quotations and quasi-quotations

The quotation name of an expression $a_1 a_2 \ldots a_n$ is the expression we get by putting the expression to be named between the quotation marks ‘ and ’. Quotation names are constant names, so it makes no sense to quantify for the letters occurring in them. It makes sense to say that ‘Lucy’ is the name of a pretty girl but it is nonsense to say that for every y, ‘Lucy’ is the name of a pretty girl.

Quotations and quasi-quotations

The <u>quotation name</u> of an expression $a_1 a_2 \ldots a_n$ is the expression we get by putting the expression to be named between the quotation marks ‘ and ’. Quotation names are constant names, so it makes no sense to quantify for the letters occurring in them. It makes sense to say that ‘Lucy’ is the name of a pretty girl but it is nonsense to say that for every y, ‘Lucy’ is the name of a pretty girl.

The <u>quasi-quotation marks</u> $\ulcorner$ and $\urcorner$ delimit schemes of metalanguage expressions where some schematic letters (A, B, $C, \ldots$) occur which can be substituted by expressions of some certain (declared) type. The items on the previous slide should be understood as e.g. for each sentence A and B, the string resulting from the concatenation of A, the sign ‘$\wedge$’ and B is a sentence again. So, distinctly from the common quotation marks, it is possible to quantify into the expressions delimited by quasi-quotation marks from the outside. It is important again that this quantification should be interpreted by substitution.

Class notation

Class notation

Instead of saying that a string a has the property F we say that a is a member of the class of F-s.

Class notation

Instead of saying that a string a has the property F we say that a is a member of the class of F-s.

If $\varphi(x)$ is a sentence with the single free variable x, then the symbol $\{x : \varphi(x)\}$ is a class abstraction and may be (loosely) read as ⌜the class of φ-s⌝.

Class notation

Instead of saying that a string a has the property F we say that a is a member of the class of F-s.

If $\varphi(x)$ is a sentence with the single free variable x, then the symbol $\{x : \varphi(x)\}$ is a class abstraction and may be (loosely) read as ⌜the class of φ-s⌝.

More exactly, the sentence ⌜a is a member of $\{x : \varphi(x)\}$⌝ means that the sentence resulting from $\varphi(x)$ by the substitution of the individual term a for the free occurrences of x (expressed as $\varphi(a)$) is true.

Class notation

Instead of saying that a string a has the property F we say that a is a member of the class of F-s.

If $\varphi(x)$ is a sentence with the single free variable x, then the symbol $\{x : \varphi(x)\}$ is a class abstraction and may be (loosely) read as ⌜the class of φ-s⌝.

More exactly, the sentence ⌜a is a member of $\{x : \varphi(x)\}$⌝ means that the sentence resulting from $\varphi(x)$ by the substitution of the individual term a for the free occurrences of x (expressed as $\varphi(a)$) is true.

We use capital letters as metalanguage variables for class abstractions and the symbol '$\in$' as an abbreviation for 'is a member of'.

Class notation

Instead of saying that a string a has the property F we say that a is a member of the class of F-s.

If $\varphi(x)$ is a sentence with the single free variable x, then the symbol $\{x : \varphi(x)\}$ is a class abstraction and may be (loosely) read as ⌜the class of φ-s⌝.

More exactly, the sentence ⌜a is a member of $\{x : \varphi(x)\}$⌝ means that the sentence resulting from $\varphi(x)$ by the substitution of the individual term a for the free occurrences of x (expressed as $\varphi(a)$) is true.

We use capital letters as metalanguage variables for class abstractions and the symbol '$\in$' as an abbreviation for 'is a member of'.

Some trivial notational conventions:

Class notation

Instead of saying that a string a has the property F we say that a is a member of the class of F-s.

If $\varphi(x)$ is a sentence with the single free variable x, then the symbol $\{x : \varphi(x)\}$ is a class abstraction and may be (loosely) read as $\ulcorner$the class of φ-s$\urcorner$.

More exactly, the sentence $\ulcorner a$ is a member of $\{x : \varphi(x)\}\urcorner$ means that the sentence resulting from $\varphi(x)$ by the substitution of the individual term a for the free occurrences of x (expressed as $\varphi(a)$) is true.

We use capital letters as metalanguage variables for class abstractions and the symbol '$\in$' as an abbreviation for 'is a member of'.

Some trivial notational conventions:

- $a \notin A$

Class notation

Instead of saying that a string a has the property F we say that a is a member of the class of F-s.

If $\varphi(x)$ is a sentence with the single free variable x, then the symbol $\{x : \varphi(x)\}$ is a class abstraction and may be (loosely) read as ⌜the class of φ-s⌝.

More exactly, the sentence ⌜a is a member of $\{x : \varphi(x)\}$⌝ means that the sentence resulting from $\varphi(x)$ by the substitution of the individual term a for the free occurrences of x (expressed as $\varphi(a)$) is true.

We use capital letters as metalanguage variables for class abstractions and the symbol '∈' as an abbreviation for 'is a member of'.

Some trivial notational conventions:

- $a \notin A$
- $a_1,\ a_2,\ \ldots a_n \in A$

Class notation

Instead of saying that a string a has the property F we say that a is a member of the class of F-s.

If $\varphi(x)$ is a sentence with the single free variable x, then the symbol $\{x : \varphi(x)\}$ is a class abstraction and may be (loosely) read as ⌜the class of φ-s⌝.

More exactly, the sentence ⌜a is a member of $\{x : \varphi(x)\}$⌝ means that the sentence resulting from $\varphi(x)$ by the substitution of the individual term a for the free occurrences of x (expressed as $\varphi(a)$) is true.

We use capital letters as metalanguage variables for class abstractions and the symbol '$\in$' as an abbreviation for 'is a member of'.

Some trivial notational conventions:

- $a \notin A$
- $a_1,\ a_2,\ \dots a_n \in A$
- $\{a_1,\ a_2,\ \dots a_n\}$

Class relations, operations, etc.

Class relations, operations, etc.

$$A \subseteq B \Leftrightarrow_{Df} \bigwedge x(x \in A \Rightarrow x \in B) \qquad \text{(Subset)}$$

Class relations, operations, etc.

$$A \subseteq B \Leftrightarrow_{Df} \bigwedge x(x \in A \Rightarrow x \in B) \qquad \text{(Subset)}$$

$$A = B \Leftrightarrow_{Df} (A \subseteq B \wedge B \subseteq A) \qquad \text{(Class identity)}$$

Class relations, operations, etc.

$$A \subseteq B \Leftrightarrow_{Df} \bigwedge x(x \in A \Rightarrow x \in B) \qquad \text{(Subset)}$$

$$A = B \Leftrightarrow_{Df} (A \subseteq B \wedge B \subseteq A) \qquad \text{(Class identity)}$$

$$A \subset B \Leftrightarrow_{Df} (A \subseteq B \wedge A \neq B) \qquad \text{(Proper subclass)}$$

Class relations, operations, etc.

$$A \subseteq B \Leftrightarrow_{Df} \bigwedge x(x \in A \Rightarrow x \in B) \qquad \text{(Subset)}$$

$$A = B \Leftrightarrow_{Df} (A \subseteq B \land B \subseteq A) \qquad \text{(Class identity)}$$

$$A \subset B \Leftrightarrow_{Df} (A \subseteq B \land A \neq B) \qquad \text{(Proper subclass)}$$

$$\emptyset =_{Df} \{x : x \neq x\} \qquad \text{(Empty class)}$$

Class relations, operations, etc.

$$A \subseteq B \Leftrightarrow_{Df} \bigwedge x(x \in A \Rightarrow x \in B) \qquad \text{(Subset)}$$

$$A = B \Leftrightarrow_{Df} (A \subseteq B \wedge B \subseteq A) \qquad \text{(Class identity)}$$

$$A \subset B \Leftrightarrow_{Df} (A \subseteq B \wedge A \neq B) \qquad \text{(Proper subclass)}$$

$$\emptyset =_{Df} \{x : x \neq x\} \qquad \text{(Empty class)}$$

$$A \cup B =_{Df} \{x : x \in A \vee x \in B\} \qquad \text{(Union)}$$

Class relations, operations, etc.

$$A \subseteq B \Leftrightarrow_{Df} \bigwedge x(x \in A \Rightarrow x \in B) \qquad \text{(Subset)}$$

$$A = B \Leftrightarrow_{Df} (A \subseteq B \wedge B \subseteq A) \qquad \text{(Class identity)}$$

$$A \subset B \Leftrightarrow_{Df} (A \subseteq B \wedge A \neq B) \qquad \text{(Proper subclass)}$$

$$\emptyset =_{Df} \{x : x \neq x\} \qquad \text{(Empty class)}$$

$$A \cup B =_{Df} \{x : x \in A \vee x \in B\} \qquad \text{(Union)}$$

$$A \cap B =_{Df} \{x : x \in A \wedge x \in B\} \qquad \text{(Intersection)}$$

Class relations, operations, etc.

$$A \subseteq B \Leftrightarrow_{Df} \bigwedge x(x \in A \Rightarrow x \in B) \qquad \text{(Subset)}$$

$$A = B \Leftrightarrow_{Df} (A \subseteq B \wedge B \subseteq A) \qquad \text{(Class identity)}$$

$$A \subset B \Leftrightarrow_{Df} (A \subseteq B \wedge A \neq B) \qquad \text{(Proper subclass)}$$

$$\emptyset =_{Df} \{x : x \neq x\} \qquad \text{(Empty class)}$$

$$A \cup B =_{Df} \{x : x \in A \vee x \in B\} \qquad \text{(Union)}$$

$$A \cap B =_{Df} \{x : x \in A \wedge x \in B\} \qquad \text{(Intersection)}$$

$$A - B =_{Df} \{x : x \in A \wedge x \notin B\} \qquad \text{(Difference)}$$

Class relations, operations, etc.

$$A \subseteq B \Leftrightarrow_{Df} \bigwedge x(x \in A \Rightarrow x \in B) \qquad \text{(Subset)}$$

$$A = B \Leftrightarrow_{Df} (A \subseteq B \wedge B \subseteq A) \qquad \text{(Class identity)}$$

$$A \subset B \Leftrightarrow_{Df} (A \subseteq B \wedge A \neq B) \qquad \text{(Proper subclass)}$$

$$\emptyset =_{Df} \{x : x \neq x\} \qquad \text{(Empty class)}$$

$$A \cup B =_{Df} \{x : x \in A \vee x \in B\} \qquad \text{(Union)}$$

$$A \cap B =_{Df} \{x : x \in A \wedge x \in B\} \qquad \text{(Intersection)}$$

$$A - B =_{Df} \{x : x \in A \wedge x \notin B\} \qquad \text{(Difference)}$$

$$A \text{ is disjoint from } B \Longleftrightarrow_{Df} A \cap B = \emptyset$$

Class relations, operations, etc.

$$A \subseteq B \Leftrightarrow_{Df} \bigwedge x(x \in A \Rightarrow x \in B) \qquad \text{(Subset)}$$

$$A = B \Leftrightarrow_{Df} (A \subseteq B \wedge B \subseteq A) \qquad \text{(Class identity)}$$

$$A \subset B \Leftrightarrow_{Df} (A \subseteq B \wedge A \neq B) \qquad \text{(Proper subclass)}$$

$$\emptyset =_{Df} \{x : x \neq x\} \qquad \text{(Empty class)}$$

$$A \cup B =_{Df} \{x : x \in A \vee x \in B\} \qquad \text{(Union)}$$

$$A \cap B =_{Df} \{x : x \in A \wedge x \in B\} \qquad \text{(Intersection)}$$

$$A - B =_{Df} \{x : x \in A \wedge x \notin B\} \qquad \text{(Difference)}$$

$$A \text{ is disjoint from } B \Longleftrightarrow_{Df} A \cap B = \emptyset$$

This is the end of the enumeration of our metalanguage tools.

Language radices

Language radices

Objects of our theory will be languages. A language is characterized:

Language radices

Objects of our theory will be languages. A language is characterized:

1. By its alphabet, i.e. a finite collection $\mathcal{A}$ of objects called letters;

Language radices

Objects of our theory will be languages. A language is characterized:

1. By its alphabet, i.e. a finite collection $\mathcal{A}$ of objects called letters;
2. By the set $\mathcal{A}^\circ$ of its strings or words;

Language radices

Objects of our theory will be languages. A language is characterized:

1. By its alphabet, i.e. a finite collection $\mathcal{A}$ of objects called letters;
2. By the set $\mathcal{A}^\circ$ of its strings or words;
3. By the operation $^\frown$ of concatenation over the strings;

Language radices

Objects of our theory will be languages. A language is characterized:

1. By its alphabet, i.e. a finite collection $\mathcal{A}$ of objects called letters;
2. By the set $\mathcal{A}^{\circ}$ of its strings or words;
3. By the operation $^\frown$ of concatenation over the strings;
4. By the fact that there is a neutral element for the operation of concatenation: the empty word $\varnothing$;

Language radices

Objects of our theory will be languages. A language is characterized:

1. By its alphabet, i.e. a finite collection $\mathcal{A}$ of objects called letters;
2. By the set $\mathcal{A}^\circ$ of its strings or words;
3. By the operation $^\frown$ of concatenation over the strings;
4. By the fact that there is a neutral element for the operation of concatenation: the empty word $\varnothing$;
5. By various classes of strings regarded as categories of meaningful expressions.

Language radices

Objects of our theory will be languages. A language is characterized:

1. By its alphabet, i.e. a finite collection $\mathcal{A}$ of objects called letters;
2. By the set $\mathcal{A}^\circ$ of its strings or words;
3. By the operation $^\frown$ of concatenation over the strings;
4. By the fact that there is a neutral element for the operation of concatenation: the empty word $\varnothing$;
5. By various classes of strings regarded as categories of meaningful expressions.

The first four items are together the radix of the language. We want to describe it axiomatically because we don't want to refer to set theory for making precise the concepts (finiteness etc.) used in the above enumeration.

Axioms for language radices

Axioms for language radices

$$\mathcal{A} \subseteq \mathcal{A}^\circ \text{ and } \varnothing \in \mathcal{A}^\circ \tag{R1}$$

Axioms for language radices

$$\mathcal{A} \subseteq \mathcal{A}^\circ \text{ and } \varnothing \in \mathcal{A}^\circ \tag{R1}$$

$$x, y \in \mathcal{A}^\circ \Rightarrow x^\frown y \in \mathcal{A}^\circ \tag{R2}$$

Axioms for language radices

$$\mathcal{A} \subseteq \mathcal{A}^{\circ} \text{ and } \varnothing \in \mathcal{A}^{\circ} \tag{R1}$$

$$x, y \in \mathcal{A}^{\circ} \Rightarrow x^{\frown} y \in \mathcal{A}^{\circ} \tag{R2}$$

$$x, y, z \in \mathcal{A}^{\circ} \Rightarrow (x^{\frown} y)^{\frown} z = x^{\frown}(y^{\frown} z) \tag{R3}$$

Axioms for language radices

$$\mathcal{A} \subseteq \mathcal{A}^\circ \text{ and } \varnothing \in \mathcal{A}^\circ \tag{R1}$$

$$x, y \in \mathcal{A}^\circ \Rightarrow x^\frown y \in \mathcal{A}^\circ \tag{R2}$$

$$x, y, z \in \mathcal{A}^\circ \Rightarrow (x^\frown y)^\frown z = x^\frown (y^\frown z) \tag{R3}$$

Let us leave off the antecedent $x(, y, \text{ etc.}) \in \mathcal{A}^\circ$; Latin lowercase letters are always variables for strings.

Axioms for language radices

$$\mathcal{A} \subseteq \mathcal{A}^\circ \text{ and } \varnothing \in \mathcal{A}^\circ \tag{R1}$$

$$x, y \in \mathcal{A}^\circ \Rightarrow x^\frown y \in \mathcal{A}^\circ \tag{R2}$$

$$x, y, z \in \mathcal{A}^\circ \Rightarrow (x^\frown y)^\frown z = x^\frown (y^\frown z) \tag{R3}$$

Let us leave off the antecedent $x(, y, \text{ etc.}) \in \mathcal{A}^\circ$; Latin lowercase letters are always variables for strings.

$$x \neq \varnothing \Leftrightarrow \bigvee y \bigvee \alpha (\alpha \in \mathcal{A} \wedge x = y^\frown \alpha) \tag{R4}$$

$$(\alpha, \beta \in \mathcal{A} \wedge x^\frown \alpha = y^\frown \beta) \Rightarrow (x = y \wedge \alpha = \beta) \tag{R5}$$

Axioms for language radices

$$\mathcal{A} \subseteq \mathcal{A}^\circ \text{ and } \varnothing \in \mathcal{A}^\circ \tag{R1}$$

$$x, y \in \mathcal{A}^\circ \Rightarrow x^\frown y \in \mathcal{A}^\circ \tag{R2}$$

$$x, y, z \in \mathcal{A}^\circ \Rightarrow (x^\frown y)^\frown z = x^\frown (y^\frown z) \tag{R3}$$

Let us leave off the antecedent $x(, y,$ etc.$) \in \mathcal{A}^\circ$; Latin lowercase letters are always variables for strings.

$$x \neq \varnothing \Leftrightarrow \bigvee y \bigvee \alpha(\alpha \in \mathcal{A} \wedge x = y^\frown \alpha) \tag{R4}$$

$$(\alpha, \beta \in \mathcal{A} \wedge x^\frown \alpha = y^\frown \beta) \Rightarrow (x = y \wedge \alpha = \beta) \tag{R5}$$

$$(x^\frown y = x \Leftrightarrow y = \varnothing) \wedge ((x^\frown y = y \Leftrightarrow x = \varnothing)) \tag{R6}$$

Axioms for language radices

$$\mathcal{A} \subseteq \mathcal{A}^\circ \text{ and } \varnothing \in \mathcal{A}^\circ \tag{R1}$$

$$x, y \in \mathcal{A}^\circ \Rightarrow x^\frown y \in \mathcal{A}^\circ \tag{R2}$$

$$x, y, z \in \mathcal{A}^\circ \Rightarrow (x^\frown y)^\frown z = x^\frown (y^\frown z) \tag{R3}$$

Let us leave off the antecedent $x(, y,$ etc.$) \in \mathcal{A}^\circ$; Latin lowercase letters are always variables for strings.

$$x \neq \varnothing \Leftrightarrow \bigvee y \bigvee \alpha(\alpha \in \mathcal{A} \wedge x = y^\frown \alpha) \tag{R4}$$

$$(\alpha, \beta \in \mathcal{A} \wedge x^\frown \alpha = y^\frown \beta) \Rightarrow (x = y \wedge \alpha = \beta) \tag{R5}$$

$$(x^\frown y = x \Leftrightarrow y = \varnothing) \wedge ((x^\frown y = y \Leftrightarrow x = \varnothing)) \tag{R6}$$

$$\text{That's all.} \tag{R7}$$

On alphabets

If our alphabet consists of the first two numerals, then we should write $\mathcal{A} = \{\text{`0'}, \text{`1'}\}$. We may have two „ideologies” for omitting the quotation marks in such cases:

On alphabets

If our alphabet consists of the first two numerals, then we should write $\mathcal{A} = \{‘0’, ‘1’\}$. We may have two „ideologies” for omitting the quotation marks in such cases:

- We use letters *autonymously*, i. e. as their own names.

On alphabets

If our alphabet consists of the first two numerals, then we should write $\mathcal{A} = \{\text{`0'}, \text{`1'}\}$. We may have two „ideologies" for omitting the quotation marks in such cases:

- We use letters *autonymously*, i. e. as their own names.
- We don't need to know what the letters are; it is enough that we have names for them.

On alphabets

If our alphabet consists of the first two numerals, then we should write $\mathcal{A} = \{\text{‘0’}, \text{‘1’}\}$. We may have two „ideologies" for omitting the quotation marks in such cases:

- We use letters *autonymously*, i. e. as their own names.
- We don't need to know what the letters are; it is enough that we have names for them.

Consider first the one-letter alphabet

$$\mathcal{A}_0 = \{\alpha\}$$

This is sufficient to name the natural numbers.

On alphabets

If our alphabet consists of the first two numerals, then we should write $\mathcal{A} = \{`0', `1'\}$. We may have two „ideologies" for omitting the quotation marks in such cases:

- We use letters *autonymously*, i. e. as their own names.
- We don't need to know what the letters are; it is enough that we have names for them.

Consider first the one-letter alphabet

$$\mathcal{A}_0 = \{\alpha\}$$

This is sufficient to name the natural numbers.
The two-letter alphabet

$$\mathcal{A}_1 = \{\alpha, \beta\}$$

is sufficient for anything.

On alphabets

If our alphabet consists of the first two numerals, then we should write $\mathcal{A} = \{`0', `1'\}$. We may have two „ideologies" for omitting the quotation marks in such cases:

- We use letters *autonymously*, i. e. as their own names.
- We don't need to know what the letters are; it is enough that we have names for them.

Consider first the one-letter alphabet

$$\mathcal{A}_0 = \{\alpha\}$$

This is sufficient to name the natural numbers.
The two-letter alphabet

$$\mathcal{A}_1 = \{\alpha, \beta\}$$

is sufficient for anything.
I. e., any language over some finite alphabet can be simulated by $\mathcal{A}_1$.

Inductive definitions of string classes

Inductive definitions of string classes

Let $\mathcal{C}$ be a finite alphabet and $\mathcal{C}^{\circ}$ the class of the strings over it. The inductive definition of an F subclass of $\mathcal{C}^{\circ}$ consists of the following three components:

Inductive definitions of string classes

Let $\mathcal{C}$ be a finite alphabet and $\mathcal{C}^\circ$ the class of the strings over it. The inductive definition of an F subclass of $\mathcal{C}^\circ$ consists of the following three components:

- Base of the induction: a class $B \subseteq \mathcal{C}^\circ$ given by some definition. We stipulate that $B \subseteq F$.

Inductive definitions of string classes

Let $\mathcal{C}$ be a finite alphabet and $\mathcal{C}^{\circ}$ the class of the strings over it. The inductive definition of an F subclass of $\mathcal{C}^{\circ}$ consists of the following three components:

- Base of the induction: a class $B \subseteq \mathcal{C}^{\circ}$ given by some definition. We stipulate that $B \subseteq F$.
- Inductive rules: a finite collection of stipulations of the form

$$\ulcorner a_1, a_2, \ldots, a_n \in F \Rightarrow b \in F \urcorner$$

where $a_1, \ldots a_n, b$ are strings over an alphabet $\mathcal{C} \cup \mathcal{V}$. (The members of $\mathcal{V}$ are understood as variables over $\mathcal{C}^{\circ}$.)

Inductive definitions of string classes

Let $\mathcal{C}$ be a finite alphabet and $\mathcal{C}^\circ$ the class of the strings over it. The inductive definition of an F subclass of $\mathcal{C}^\circ$ consists of the following three components:

- Base of the induction: a class $B \subseteq \mathcal{C}^\circ$ given by some definition. We stipulate that $B \subseteq F$.
- Inductive rules: a finite collection of stipulations of the form

$$\ulcorner a_1, a_2, \ldots, a_n \in F \Rightarrow b \in F \urcorner$$

  where $a_1, \ldots a_n, b$ are strings over an alphabet $\mathcal{C} \cup \mathcal{V}$. (The members of $\mathcal{V}$ are understood as variables over $\mathcal{C}^\circ$.)
- Closure: the members of F are just the strings produced from the basis by finitely many applications of the inductive rules.

Inductive definitions of string classes

Let $\mathcal{C}$ be a finite alphabet and $\mathcal{C}^\circ$ the class of the strings over it. The inductive definition of an F subclass of $\mathcal{C}^\circ$ consists of the following three components:

- Base of the induction: a class $B \subseteq \mathcal{C}^\circ$ given by some definition. We stipulate that $B \subseteq F$.
- Inductive rules: a finite collection of stipulations of the form

$$\ulcorner a_1, a_2, \ldots, a_n \in F \Rightarrow b \in F \urcorner$$

  where $a_1, \ldots a_n, b$ are strings over an alphabet $\mathcal{C} \cup \mathcal{V}$. (The members of $\mathcal{V}$ are understood as variables over $\mathcal{C}^\circ$.)
- Closure: the members of F are just the strings produced from the basis by finitely many applications of the inductive rules.

We assume that the closure condition works (and we don't mention it any more).