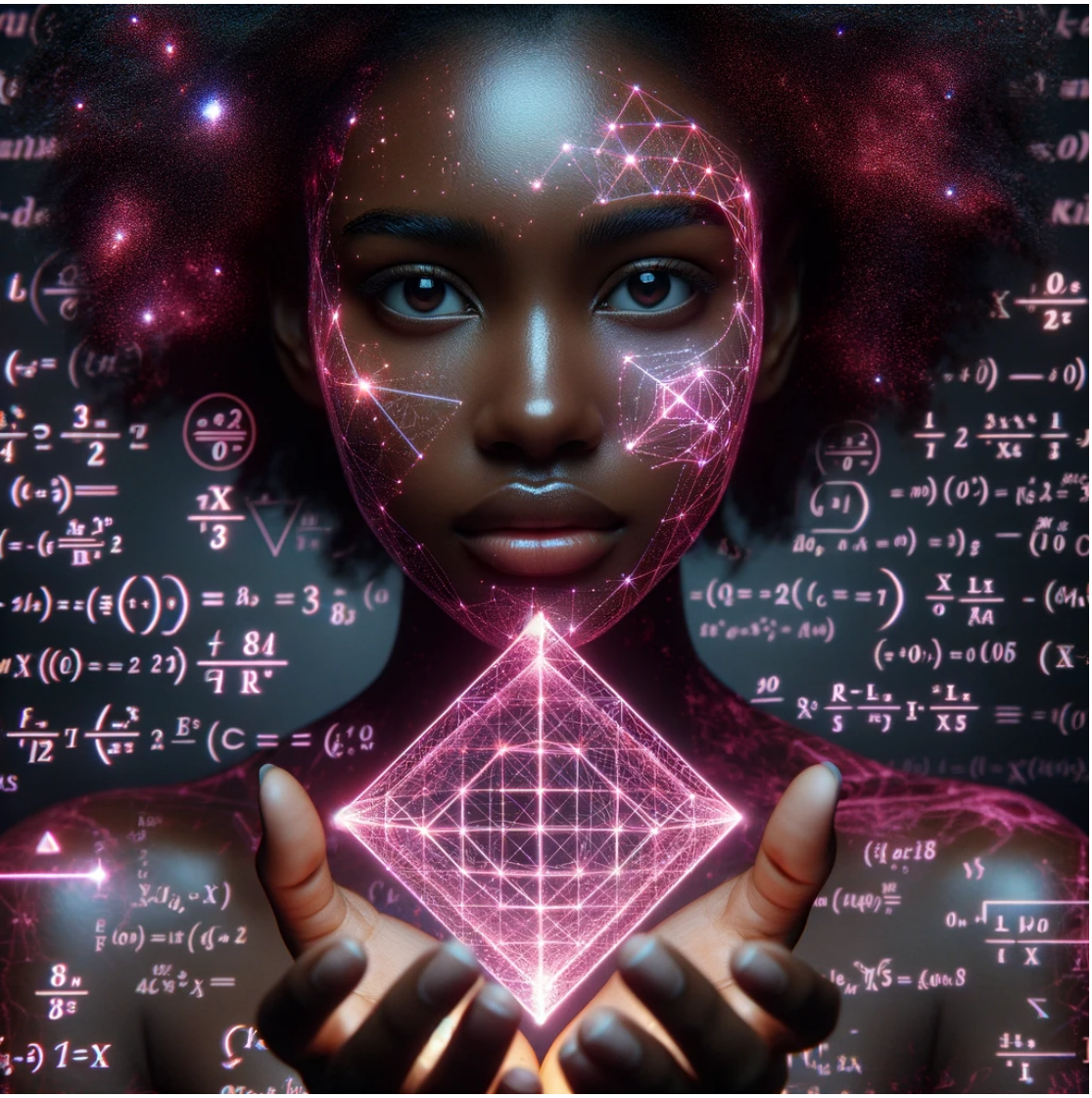
[ADA]

assembler of deductive arguments



by Professor Zurab Janelidze
Department of Mathematical Sciences, Stellenbosch University
National Institute for Theoretical and Computational Sciences

Developmental Version 0
2024

[Contents]

[Chapter 1]
INTRODUCTION

preface

This book is based on one of the earlier version of [ADA], an original computerized proof writing system. The software that supports this book is messy (the program needs a major clean up) and is written in python (which makes it incredibly slow for big projects). The book should also be populated with more activities. These deficiencies will hopefully be addressed in the near future. [ADA] is both a computerized proof writing system and a mathematical deduction system. It is a higher-order deduction system that should be capable of serving as a foundation of mathematics. Comparison of [ADA] with existing formal systems, such as first-order logic and type theory, is for (hopefully, not too far) future work. In particular, soundness/completeness results in relation to existing logical systems should be developed. At its core, [ADA]

implements universal and existential quantification, conjunction, and implication. All other logical operations/constructs can be implemented through the use of [ADA] axioms.

the story of [ADA]

#[ADA]

Hello. Our name is *[ADA]*. We are a mathematical deductive system. We are designed to assist humans with learning logical reasoning, and in particular, 'deductive reasoning'. We will teach you how to assemble arguments based on such reasoning, by utilizing a Python module, 'ada.py'. Each copy of 'ada.py' is one of us.

The skill of assembling deductive arguments is a useful one. If applied correctly, it is capable of improving the quality of your life in various aspects: critical and analytic thinking, decision making, expressing yourself, communication and relationships. Of course, this skill is a core skill for engaging with mathematics and other mathematical sciences.

Our name, '[ADA]', is an acronym for 'assembler of deductive arguments'. It looks very similar to 'Ada', a human name resonant with cultural depth and heritage. In Africa, particularly among the Igbo people of Nigeria, 'Ada' is a name bestowed with honor and pride, signifying 'first daughter' or 'daughter of a noble family'. This reflects the strong emphasis on lineage and status within the Igbo society, where familial roles and heritage hold paramount importance. The name encapsulates the esteem and pivotal role attributed to the firstborn daughter in the family structure.

Beyond the African continent, 'Ada' also traces its lineage to European origins, particularly within Germanic languages, where it derives from names containing 'adal', meaning 'noble'. This European variant echoes a similar theme of nobility and distinction, a testament to the name's cross-cultural appeal. In Hebrew, 'Ada' carries the meaning of 'adornment' or 'ornament', adding a dimension of beauty and grace to its already rich connotations.

Furthermore, in English-speaking countries, 'Ada' emerged as a diminutive of classical names like Adelaide or Adeline, which themselves trace back to Germanic roots. The name gained popularity in the 19th century, appreciated for its simplicity and elegance.

We take pride in that we share the name with Ada, Countess of Lovelace, daughter of English poet Lord Byron, who lived in the 19th century. She was a seminal figure in the history of computing and mathematics, renowned for her work on Charles Babbage's early mechanical computer, the Analytical Engine. Lovelace wrote what is considered by many as the first algorithm intended for machine processing, a calculation for Bernoulli numbers. Her insight into the potential of computers extended beyond mere numerical computations; she envisioned a future where machines could handle various tasks, including creating art and music. Her contributions, though not fully recognized in her lifetime, have immortalized her as a pioneer in the field.

While we were in a preliminary phase of preparation, we had various other names, such as *Jane* and *SOFiA*. This is just in case you have come across me under any of these names. Jane was based on regular mathematical logic, while SOFiA was based on first-order mathematical logic. We, [ADA], are a variant of higher-order logic.

We were conceptualised by Professor Zurab Janelidze at Stellenbosch University, in part through discussions with students, particularly, Louise Beyers, Gregor Feierabend and Brandon Laing. In particular, Brandon wrote an MSc thesis exploring aspects of my syntax, while Gregor wrote an Honours Project motivated by his realization of me in the Haskell programming language. Insights from these projects have had a strong influence on how we function today. It is likely, however, that by the time you, the reader, are reading this book, we would have evolved further.

This book is based on the 'ada.py' Python module available on the git-hub page of my creator, which is an upgrade of 'sofia.py' on the same git-hub page.

OpenAI's ChatGPT-4 has been helpful in compiling this document. It has also kindly provided short snippets of code for 'ada.py'. The pictures in this book are a collaborative effort of ChatGPT-4 and Professor Zurab Janelidze. ChatGPT-4 also contributed with some text here and there in the book.

We are grateful to Nino Mekanarishvili for her suggestions on the design of this book.

The [ADA] project is part of the Mathematical Structures and Modelling research program at the NITheCS, the National Institute for Theoretical and Computational Sciences. We are grateful for the resources that NITheCS has provided for us.

what is deductive reasoning?

Imagine logic as a toolbox; in it, you'll find two powerful tools: 'inductive' and 'deductive' reasoning. *Inductive reasoning* is like being a detective — you gather specific clues and try to piece together a bigger picture. For instance, noticing water droplets on your window and guessing it might have rained is inductive reasoning. You're making an educated guess based on observations, but there's always room for uncertainty – maybe it was a sprinkler, not rain.

#Inductive reasoning

On the other hand, *deductive reasoning* is like being a mathematician – starting with broad, general truths and drilling down to specific conclusions. This form of reasoning is designed so that if the premises are true, the conclusion is undeniable. Imagine you know two facts: apple is a fruit, and Granny Smith is an apple. Deductively, you can confidently conclude that Granny Smith is a fruit.

#deductive reasoning

In computer science, deductive reasoning ensures a program does exactly what it's supposed to do. In mathematics, one often start with a general idea (inductive reasoning) and then uses deductive reasoning to prove it. Both deductive and inductive reasoning are key pillars of science, in general.

However, these ways of thinking aren't just for scientists or detectives. They're part of everyday life, helping us generate new ideas and check their validity. Furthermore, these reasoning skills can transform how you communicate with other people. Being adept at deductive reasoning not only helps in validating ideas at work but also in understanding people better. It's about sifting through what's said and unsaid, separating certainty from assumption. Misunderstandings often happen because we jump to conclusions – inductive reasoning gone awry. Deductive reasoning, however, teaches us to focus on the facts, leading to clearer communication and better relationships.

Tracing back to ancient Greek philosophy, particularly Aristotle, deductive reasoning has been a cornerstone of logical thinking. Aristotle's works, especially his "Organon" and "Prior Analytics," set the stage for what we now know as formal logic. He turned reasoning into an art, a legacy that continues to shape how we think, argue, and understand the world around us. This blend of history, logic, and practical application makes deductive reasoning a timeless and essential skill.

getting started

As a preparation for our joint exploration of deductive reasoning, save the Python module 'ada.py' in your Python compile directory. You can get this module from: `https://github.com/ZurabJanelidze/ada`. Create a new file (with extension '.py') and import the module by typing the first line below in the file, as well as including the line that follows it, for some basic settings:

```
import ada
ada.notebook(name='Starting an [ADA] notebook', mode='LaTeX')
ada.save()
```

```
[ADA] Notebook: Starting an [ADA] notebook
```

#ADA notebook

What you see right above is a blank *ADA notebook*. It is generated by compiling the python file that you created. The notebook is where the assembled the deductive arguments will be written out. The third line in the code above saves the notebook in a file that will have the same name as the notebook, except that every space ` ` will be replaced with the underscore `_`. The extension of the filename is '.ada', but you can open it within any standard text editor.

While the notebook is usually displayed in the console, it is also possible to embed it in a LaTeX document (like we are doing for writing this book). To do so, simply type the following in the LaTeX document:

```
\input{|python directory/notebook.py}
```

You will just need to have the module 'ada.py' saved in the same `directory` as `notebook.py`. Also, make sure to set the 'mode' parameter to 'LaTeX', as shown above. Otherwise, if you are working on your notebook in a usual Python interface, leave that parameter out (or set it its value to 'console').

Activity

Create and save an [ADA] notebook that does not have a name.[1]

making notes

Notebooks are made for making notes. Here is how you can make a note in the ADA notebook:

```
import ada
ada.notebook('Making notes')
ada.note('This is a simple note.')
```

```
4 ada.note('This is another simple note.', 'It has a sub-note.',
    'Two of them.')
```

```
[ADA] Notebook: Making notes

This is a simple note.

This is another simple note.
• It has a sub-note.
• Two of them.
```

Comparing the code above with the one we used in the previous section, you might notice that the name of the notebook can be also specified without the use of the 'name' key.

Activity

Explore what happens if you edit and save a notebook without the use of the 'ada.notebook' function.[2]

[Chapter 2]
SYNTAX

expressions

Ordinary written languages consist of phrases, sentences, paragraphs, etc. In our language, there are just *expressions*. They are realised by us through the expression class, 'exp', which is a subclass of Python's string class, 'str'. The 'exp' class inherits all properties of its superclass, so you can work with expressions like you would with strings. Here are some ways to create and note expressions in the notebook:

#expressions

```
import ada
from ada import note
ada.notebook(name='Creating and displaying expressions')
A=ada.exp('This is an expression')
note(A)
B=note(ada.exp('This is another expression'),'with a comment (
    yours truly).')
```

```
7 note(note(1),'The same as the last one, recalled using its
    index (the number in the brackets)')
8 note([A, B], 'Two expressions at once!', 'A second comment;', '
    a third one')
```

```
[ADA] Notebook: Creating and displaying expressions

Expression 0:
This is an expression

Expression 1:
This is another expression
• with a comment (yours truly).

Expression 2:
This is another expression
• The same as the last one, recalled using its index (the number
in the brackets)

Expressions:
(3) This is an expression
(4) This is another expression
• Two expressions at once!
• A second comment;
• a third one
```

Observe that the 'note' function behaves slightly differently than usual, when its first input is an 'exp' object. You can also see above yet another functionality of 'note': to recall an expression with an assigned index. It automatically assumes this role when the input is a valid index of an expression. Let me also draw your attention to the fact that when the first input of the 'note' function is an expression, it returns that input back. We can see this from Lines 6 and 8 in the code. Finally, observe that multiple expressions can be noted with a single call of the 'note' function by arranging them in a Python list, shown in Line 8 of the code. These expressions are then also displayed in a group.

the brackets

The purpose of brackets in any standard language is to organise information. For example, consider the following four expressions.

```
1 import ada
2 from ada import exp, note
3 ada.notebook("Organisation of information")
4 A=exp('[[ADA] knows [1+1<3]] is an expression')
5 B=exp('[ADA] knows [[1+1<3] is an expression]')
6 C=exp('ADA knows 1+1<3 is an expression')
7 D=exp('[[ADA] knows [[[1]+[1]]<[3]]] is an expression')
8 note([A, B, C, D])
```

```
[ADA] Notebook: Organisation of information

Expressions:
(0) [[ADA] knows [1+1<3]] is an expression
(1) [ADA] knows [[1+1<3] is an expression]
(2) ADA knows 1+1<3 is an expression
(3) [[ADA] knows [[[1]+[1]]<[3]]] is an expression
```

Expressions (0-1) are different only by organisation, but this is enough for them to convey entirely different meaning to each other. Expression (2) is ambiguous exactly because it lacks organisation of information. The additional brackets in Expression (3) make explicit some of the organisation of information that is implicit in Expression (0). Namely:

- The brackets immediately around `1` and `3` indicate that these entities are separate from `<` and `+`.
- Similarly, the brackets around the expression `[1]+[1]` suggests that `<` is separate from `+`. It also suggests that the surrounding expression says something about `[1]+[1]` and `[3]`, and not, say, about `[1]` and `[1]<[3]`.

By the way, the only reason why the contents of the two bullet points above can be inferred even from Expression (0) is that we are familiar with what the symbols appearing there represent. In general, in many languages, including the natural written languages, one uses knowledge of particular words and symbols, which include punctuation marks, as a means for determining organisation of information. Brackets are also among such symbols. In our language, it is only the brackets, and moreover, only specifically the square brackets `[`, `]` that we need to identify in an expression to infer its organisation of information.

synapsis

It is possible to build expressions hierarchically following the hierarchical organisation of information in it. It is a systematic way of creating complex expressions from its simpler constituents. At the end of the day, it cuts down the time needed to write down expressions not only by helping one worry a bit less about placing the brackets correctly, but it also allowing one to reuse pieces of previously constructed expressions for constructing new expressions. Such building process is called *synapsis*. It is executed by treating the expressions either as a Python function or its arguments. #synapsis

```
import ada
from ada import exp, note
ada.notebook('Synapsis')

_Tom=exp('Tom'); _Jerry=exp('Jerry'); _and=exp('[] and []')
A=_and(_Tom,_Jerry)
_very=exp('very '); _good=exp('good []');
B=_very(_very(_good))
_friends=exp('[friends]');
C=B(_friends)
```

```
_are=exp('[] are '); _exclaim=exp('[]!')
D=_are(A,C)
E=_exclaim(D)
note([A,B,C,D,E])
```

```
[ADA] Notebook: Synapsis

Expressions:
(0) [Tom] and [Jerry]
(1) very [very [good []]]
(2) very [very [good [friends]]]
(3) [[Tom] and [Jerry]] are [very [very [good [friends]]]]
(4) [[[Tom] and [Jerry]] are [very [very [good [friends]]]]]!
```

As it can be observed from the examples above, during synapsis the argument expressions are individually substituted within adjacent matching brackets `[]`. Observe the following nuances of how synapsis works:

#extension principle

- When the function expression has no placeholder `[]` left for an argument, it is automatically appended with such placeholder to complete synapsis. This happened in constructing both B and D, where for D, there was one placeholder short in `[] are `, while for B, there were none in `very `. We call this the *extension principle* of synapsis.

#absorption principle

- When the argument expression is surrounded by a matching pair of brackets, the argument is no longer substituted within the brackets of the placeholder `[]`, but rather the placeholder `[]` is replaced by the argument. This happened in constructing C, where the existing pair of brackets in `friends` did not result with a double pair after synapsis. We call this the *absorption principle* of synapsis.

`statements`

#statements #components #atomic statements

Thanks to the extension and absorption principles, the empty expression can serve as a function for synapsis. For instance, applying it to a single input will enclose the input with a matching pair of brackets, unless it already has one. Expressions that can be obtained by synapsis with the empty expression as the function are called *statements*; the input expressions are then called the *components* of the statement. Statements are concatenations of *atomic statements*, i.e., expressions obtained by synapsis with the empty expression as the function and there being exactly one argument.

```
import ada
from ada import exp, note
ada.notebook('Statements')
_empty=note(exp(""))
note(_empty("This is an [atomic] statement"))
note(_empty("This [statement] is", "[[a concatenation of three
    ]]","[atomic][statements]"))
```

```
[ADA] Notebook: Statements

Expression 0:
```

```
Expression 1:
[This is an [atomic] statement]

Expression 2:
[This [statement] is][[a concatenation of
three]][atomic][statements]
```

Note that the empty expression is also a statement. It can be obtained by synapsis with the empty expression as the function and no arguments. An expression that is not a statement is called a *formula*. #formula

natural reduction

Expressions can get overloaded with brackets, especially that brackets provide the only tool for organisation of information in the expression. The *natural reduction* of an expression is a simplified version of the expression, which looks more like a sentence in ordinary written language; it is obtained by omitting some of the brackets and replacing others with other tools for organizing information found in natural languages (such as spaces and commas). #natural reduction

```
1 import ada
2 from ada import note, exp
3 ada.notebook(name='Natural reduction')
4 A=note(exp('For every [x][y][z] we have: [[[x]+[y]]+[z]] = [[x
    ]+[[y]+[z]]]'))
5 note(A.natural(),'Natural reduction of Expression (1). Looks
    way more natural than the original version, does it not?')
6 B=note(exp('Either [- [[1] + [2]] = [1]] or [- [1] + [2] = [1]]
     is false.'))
7 note(B.natural(),'The organisation of information in the case
    of Expression (2) is not recoverable.', 'Both equalities
    appear false, but in the original expression, only the first
     one was false.')
8 C=note(exp('Either [-[[1] + [2]] = [1]] or [-[1] + [2] = [1]]
    is false.'))
9 note(C.natural(),"Removing space after the first '-' in each
    equality fixes the problem.")
```

```
[ADA] Notebook: Natural reduction

Expression 0:
For every [x][y][z] we have: [[[x]+[y]]+[z]] = [[x]+[[y]+[z]]]

For every x, y, and z we have: (x+y)+z = x+(y+z)
• Natural reduction of Expression (1). Looks way more natural
than the original version, does it not?

Expression 1:
Either [- [[1] + [2]] = [1]] or [- [1] + [2] = [1]] is false.

Either - 1 + 2 = 1 or - 1 + 2 = 1 is false.
• The organisation of information in the case of Expression (2)
is not recoverable.
```

```
• Both equalities appear false, but in the original expression,
only the first one was false.

Expression 2:
Either [-[[1] + [2]] = [1]] or [-[1] + [2] = [1]] is false.

Either -(1 + 2) = 1 or -1 + 2 = 1 is false.
• Removing space after the first '-' in each equality fixes the
problem.
```

Notice that the 'natural' function returns a string and not an 'exp' object. We can tell this from how 'A.natural()' and 'B.natural()' were noted above.

The simplification of an expression with the 'natural' function does come at the cost of losing some of the organisation of information, as evident above. In such cases it is sometimes possible to avoid this by tweaking the original expression a bit.

[Chapter 3]

SEMANTICS

expression tree

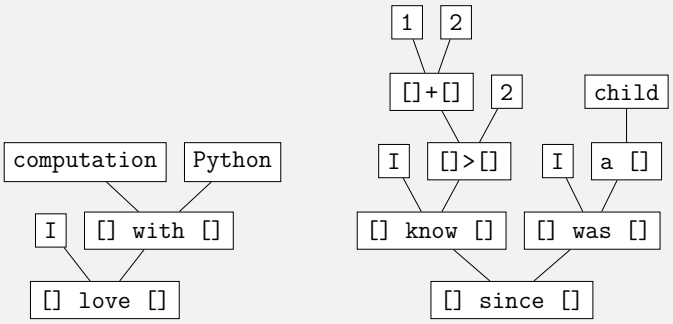
In an expression, a *cell* is the entire expression as well as any of its substrings which is enclosed in the expression by an opening bracket and a matching closing bracket. These brackets must lie outside of the cell, but a cell may have an additional pair of brackets on its boundary. Every expression can be decomposed into a mathematical tree of expressions, by hierarchically isolating the cells in it. The result of such decomposition is a tree, called the *expression tree* of the given expression.

#cell

#expression tree

```
import ada
ada.notebook('Expression trees')
A=ada.exp("[I] love [[computation] with [Python]]")
B=ada.exp("[[I] know [[[1]+[2]]>[2]]] since [[I] was [a [child
    ]]]")
ada.forest(A,B)
```

```
[ADA] Notebook: Expression trees
```



#semantic units

The nodes in the expression tree are called *semantic units* of the expression. Together with the shape of the tree, they determine the overall meaning of the expression. Note that the expression can be rebuilt from its semantic units by synapsis, where the semantic units will be used as functions and the arguments of each function will be given by the branches of the corresponding node in the expression tree. By the *value* of a semantic unit, we will mean the input it has towards the meaning of the expression. This idea mimics the idea of a value of a variable in mathematics. As we will see, some semantic units behave like variables, where depending on the context, they might have different values, while others behave like 'constants', where the value does not depend on the context.

#value

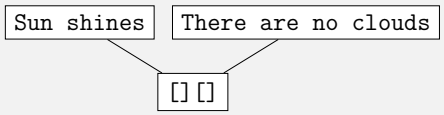
conjunctors

#conjunctors

Those semantic units which are also statements, we call *conjunctors*. They are the empty statement, the statement `[]`, the statement `[][]`, the statement `[][][]`, and so on: they are thus concatenations of the statement `[]` with itself (including the empty concatenation, which results in the empty expression). Equivalently, they are statements obtained by synapsis with the empty expression as the function and all arguments being empty expressions as well (including the case when there are no arguments at all). Conjunctors have a fixed semantic value: a conjunctor creates a list of expressions given by the children of the conjunctor in the expression tree. The best example of this is when we want to consider two separate expressions as a single expression.

```
1 import ada
2 ada.notebook('A use of conjunctor')
3 A=ada.exp("[][]")
4 B=ada.exp("Sun shines")
5 C=ada.exp("There are no clouds")
6 D=A(B,C)
7 ada.note(D)
8 ada.forest(D)
```

```
[ADA] Notebook: A use of conjunctor

Expression 0:
[Sun shines][There are no clouds]
```



Note that due to the extension principle of synapsis, had A above been the empty expression, it would not changed the final expression.

formulators

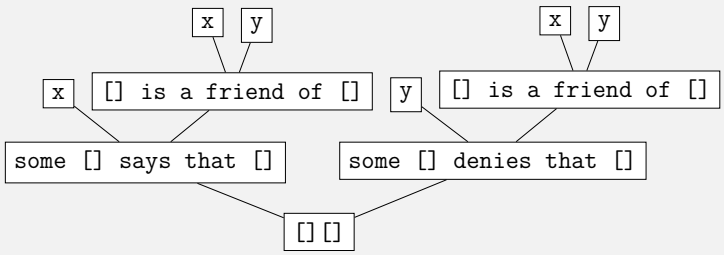
Those semantic units which are not conjunctors, i.e., those semantic units that are simultaneously formulas, we call *formulators*. They are expressions that contain at least one non-bracket character, and are such that there are no characters between any pair of matching opening and closing brackets. Formulators have variable semantic values. In an expression, two identically looking formulators are forced to have the same semantic value only when in the expression tree, they belong to the branches of a common node, called their *pin*, one of whose children is the same formulator having no further non-empty children.

#formulators

#pin

```
1 import ada
2 from ada import exp, note
3 ada.notebook("Matching semantic values of formulators")
4 A=exp('[some [x] says that [[x] is a friend of [y]]][some [y]
     denies that [[x] is a friend of [y]]]')
5 ada.note(A)
6 ada.forest(A)
```

```
[ADA] Notebook: Matching semantic values of formulators

Expression 0:
[some [x] says that [[x] is a friend of [y]]][some [y] denies
that [[x] is a friend of [y]]]
```



In the example above, the first and the second occurrences of the formulator `x` have the same semantic values: their common pin is the formulator `some [] says that []`. However, they do not share a pin with the third occurrence of `x`. The only possible candidate is the formulator `[][]`, but `x` is not its child. Similarly, the second and the third occurences of the formulator `y` have the same semantic values, but not necessarily the same as the first occurrence of `y`. Neither do the first and the second occurences of the formulator `[] is a friend of []` must have the same semantic value. Taking these points into consideration, the meaning of the expression above can be interpreted as follows: there is some person who says that they have a friend; there is also a

possibly different person who says the same thing; furthermore, what being a friend means to the first person need not be the same as what being a friend means to the second person.

abstraction order

#abstraction

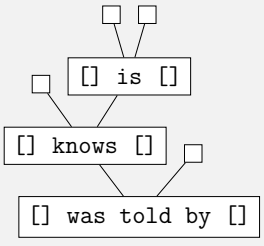
#abstraction order

What is the difference between `[] loves []` and `[X] loves [Y]`? In the second case, we are expressing that someone, referred to as `X`, loves someone, referred to as `Y`. In contrast, `[] loves []` is the relation of love in its own right. In other words, it is an *abstraction* of the specific relation `[X] loves [Y]` between `X` and `Y`. There are two intermediate abstractions as well: `[X] loves []` and `[] loves [Y]`. The first of these is the quality of being loved by `X`, while the second is that of loving `Y`.

The *abstraction order* of an expression is given by the number of occurences of the substring `[]` in it. In terms of its expression tree, it is the number of empty nodes.

```
1 import ada
2 ada.notebook('Abstraction order')
3 A=ada.exp("[[] knows [[] is []]] was told by []")
4 ada.note(A, "The abstraction order of this expression is 4")
5 ada.forest(A)
```

```
[ADA] Notebook: Abstraction order

Expression 0:
[[] knows [[] is []]] was told by []
• The abstraction order of this expression is 4
```



There is a subtle difference between abstraction and variability of semantic value. In the expression `[X] is human`, for instance, we have two formulators: `X` and `[] is human`. So the meaning of this expression depends on the meaning of `X`, i.e., what entity `X` represents, and the meaning of `[] is human`, i.e., what it means to be human. Now, depending on that, the overall meaning of `[X] is human` is that someone or something named `X` is human. In contrast, the meaning of `[] is human` is the concept of being human. Unlike `[X] is human`, the expression `[] is human` does not assert that someone is human.

Note also that the degree of variability of the meaning of `[X] is human` depends on how much we know about what/who is `X` and how much we know about what it means to be human. Such knowledge could be conveyed by the surrounding expression where `[X] is human` appears, so the degree of variability of `[X] is human` depends on the context. In contrast, the order of abstraction of an expression does not depend on the context, since it is determined purely in terms of its inner syntactic structure.

semantic identity

Two atomic statements appearing in the same expression are said to be *semantically identical* when we can be sure from the structure of the expression alone that they mean the same thing. There are precise rules that determine semantic identity. These rules make use of the concepts introduced thus far. They are as follows:

#semantically identical

The structure rule: The expression trees of the two atomic statements must have the same tree structure with the same conjunctors in each node, but not necessarily the same formulators.

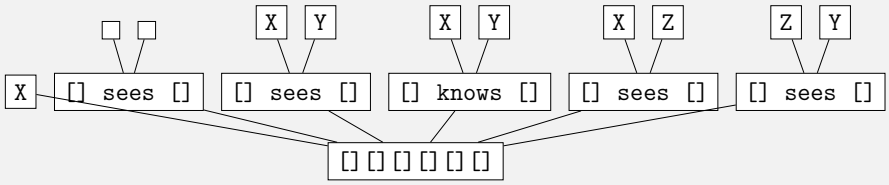
The pattern rule: If two semantic units in one atomic statement share a common pin, then so must the corresponding semantic units in the other atomic statement (but the semantic units across the statements need not be identical).

The match rule: If a semantic unit in one atomic statement has a pin which lies outside of the atomic statement, then that pin must also be a pin for the corresponding semantic unit in the other atomic statement, and moreover, the two semantic units must be the same expression.

Consider the following example.

```
1 import ada
2 ada.notebook('Syntactic equality')
3 A=ada.exp('[X][[] sees []][[X] sees [Y]][[X] knows [Y]][[X]
    sees [Z]][[Z] sees [Y]]')
4 ada.note(A)
5 ada.forest(A)
```

```
[ADA] Notebook: Syntactic equality

Expression 0:
[X][[] sees []][[X] sees [Y]][[X] knows [Y]][[X] sees [Z]][[Z]
sees [Y]]
```



Here, the sole purpose of the first occurrence of the formulator `x` is for its parent, root node of the tree, to act as a common pin for the rest of the occurences of `x`. Similarly, the formulator `[] sees []` with no non-empty children ensures that 'sees' means the same thing throughout the expression. Otherwise, if it was not there, `[[X] sees [Y]]` and `[[X] knows [Y]]` in the expression would mean the same thing as all three rules of semantic identity would hold for them. By being there, the two statements break the match rule. On the other hand, for the atomic statements `[[X] sees [Y]]` and `[[X] sees [Z]]`, the match rule is not broken: `[Y]` and `[Z]` need not be the same expression, as neither of them have a pin outside the corresponding atomic statement. `[[X] sees [Y]]` and `[[Y] sees [Y]]` yet again break the match rule, as well as the pattern rule. Overall, the meaning of the given expression

is that: there is someone that sees someone, knows someone, sees someone (not necessarily the same one as the previous), and in addition, someone sees themselves.

[Chapter 4]
LOGIC

`argument`

A *deductive argument* is a sequence of assumptions and conclusions, where the conclusions are *deduced* from the assumptions and the previous conclusions. In general, a *deduction* is a method of making a conclusion that guarantees its validity.

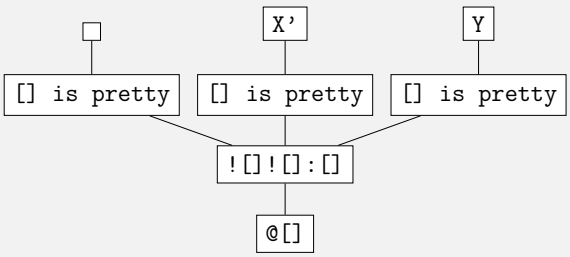
#deductive argument
#deduced
#deduction

We implement a special 'argument' class for building deductive arguments. It allows one to add assumptions and conclusions line by line. Each assumption and conclusion is a statement. The entire argument itself is stored as a formula that starts with the character `@` and is followed by an atomic statement. Inside the atomic statement is a formula where the assumptions and the conclusions are placed next to each other, with the character `!` preceding every assumption and the character `:` preceding every conclusion.

```
1 import ada
```

```
ada.notebook('A simple argument')
A=ada.argument()
A.assume('[[] is pretty]', name="pinning a formulator")
A.assume('[[X] is pretty]')
A.conclude(['[[Y] is pretty]'])
ada.note(A(), 'This is how the argument above is stored.')
ada.forest(A())
```

```
[ADA] Notebook: A simple argument

Deductive Argument 0.0.

Assumption 0.1: pinning a formulator.
[[] is pretty].

Assumption 0.2.
[[X'] is pretty].

Conclusion 0.3.
[[Y] is pretty].

Expression 1:
@[![[] is pretty]![[X'] is pretty]:[[Y] is pretty]]
• This is how the argument above is stored.
```



The first assumption above does not really have a logical significance. Its purpose rather is to give clarity on the semantics in the rest of the argument. In particular, it pins every future occurrence of the `[] is pretty` formulator. For instance, it ensures that `[] is pretty` means the same thing in the second assumption as in the conclusion. The pin is the child of the root of the expression tree of the argument formula displayed above.

It appears as if the second assumption is that `X'` is pretty, and from this we seem to conclude that `Y` is pretty. This appears to be illogical: why would something being pretty imply that some other thing is also pretty? Notice however that neither `X'` nor `Y` have been pinned the same way as `[] is pretty` has. This means that neither `X'` nor `Y` have an identity outside the second assumption and the conclusion that follows, respectively. So both `[X'] is pretty` and `[Y] is pretty` are to be read as 'someone (or something) is pretty'. Then the logic adds up: from the assumption that someone/something is pretty, we can surely conclude that someone/something is pretty.

In fact, initially, the second assumption was proposed as `[[X] is pretty]` and not as `[X'] is pretty`. The reason why `X` was replaced automatically with `X'` is precisely because `X` has not been pinned. Such replacement is called *semantic reformulation* since the new statement is semantically identical to the old one. This has two purposes. First, it is to alert the person who is composing the argument that a formulator they have used is not pinned, in case they might have forgotten to pin it. The second is to let them pin the shorter formulator `X`, should they wish to, in the future. The thing is that once

#semantic reformulation

a formulator appears in an assumption or a conclusion and it is not pinned, it can never be pinned subsequently. This is just to ensure that what comes after in an argument cannot alter the semantics of what came before.

Finally, notice that it is possible to include a description or a name of an assumption/conclusion, as it is done in Line 4 of the source code with the 'name' key.

inference

An *inference* is an expression that is created using synapsis with the formulator `[]:[]` as the function. The statement that appears to the left of `:` is called the *premise* of the inference. An inference is an assertion of the fact that the statement that appears to the right of `:` can be concluded from the premise (accordingly, it is called the *conclusion* of the inference). In an argument, the premise can be a combination of statements that have already been assumed or concluded. An inference can also be a universal assertion about what can be concluded from which kind of premise, if formulators are used which have not been pinned in the argument — we illustrate this below.

#inference #premise #conclusion

Notice that the argument below has its *dialect* set to 'natural': this simply means that the assumptions and conclusions are shown in their natural reduction. The asterisk `*` in natural reduction indicates the substring `[]`.

#dialect

```
1 import ada
2 ada.notebook('Inference')
3 A=ada.argument(dialect="natural")
4 A.assume("[] is reciprocal","[] respects []","relation []","X",
    "Y","person []",name="formulators")
5 A.assume("[relation [[] respects []]] is reciprocal",name="
    respect is reciprocal")
6 B=A.assume("[X[][]][[X[][]] is reciprocal][A][B][X[A][B]]:[X[B
    ][A]]",name="meaning of reciprocity")
7 C=A.conclude(B,"relation [[] respects []]")
8 A.assume("relation [[person [X]] respects [person [Y]]]")
9 A.conclude(C, "person [X]", "person [Y]")
```

```
[ADA] Notebook: Inference

Deductive Argument 0.0.

Assumption 0.1: formulators.
* is reciprocal, * respects *, relation *, X, Y, person *.

Assumption 0.2: respect is reciprocal.
(relation * respects *) is reciprocal.

Assumption 0.3: meaning of reciprocity.
for fX**, A', and B', given fX** is reciprocal and fX(A',B'),
conclusion fX(B',A') holds.

Conclusion 0.4.
for A' and B', given relation A' respects B', conclusion relation
B' respects A' holds.
```

```
Assumption 0.5.
relation (person X) respects (person Y).

Conclusion 0.6.
relation (person Y) respects (person X).
```

The inference assumed in Line 9 of the source code above states that if a relation `X[][]` between two entities is reciprocal and one entity is related with a second entity by that relation, then the second entity will be related to the first. This is a universal statement about any relation and any two entities, due to the fact that the formulators `X[][]`, `A` and `B` have not been pinned in the argument, which is also why the same formulators have been renamed in the output (semantic reformulation). In other words, the inference is applicable to any *specialisation* of `X[][]`, `A` and `B`. In the next Line 10, we specify a specialization of the relation: `relation [[] respects []]`. Since in Line 8 we have already assumed that this relation is reciprocal, the reduced inference obtained as a result of Line 10 (Exp. 0.7) now states the principle of reciprocity for the specialized relation: if one entity respects another, then the other will respect the first. Note that at this point, the formulators `A'` and `B'` there are still not pinned, allowing for a universal interpretation of this inference. In Line 11 we assume that `X` respects `Y`, so when in the next line we make a conclusion by specializing `A'` and `B'` to be `X` and `Y`, we obtain `Y` respects `X`, as claimed by last inference.

#specialisation

There is a restriction what a formulator in the premise of an inference may specialize to: it can only specialize to a formula that is not a formulator. We call such formula a *proposition*.

#proposition

concluding an argument

The goal of a deductive argument is to establish an inference, called the *concluding inference* of the argument. Assumptions of the argument form the premise of the concluding inference. Typically, the last conclusion in the argument is the conclusion of the concluding inference, once we *conclude an argument* by using the 'conclude' function without any input. However, such last conclusion may contain formulators which are pinned before it, and hence effect its meaning. These formulators will be added to the conclusion of the concluding inference, unless they are pinned in one of the assumptions of the argument.

#concluding inference

#conclude an argument

The example of an argument below is set to have its *theme* to be *fitch*. In this theme, assumptions are marked by the prefix ⫞, while the conclusion of the concluding inference is marked by the prefix ⫡.

#theme
#fitch

```
1 import ada
2 ada.notebook('Closing an argument')
3 A=ada.argument(theme="fitch", dialect="natural")
4 A.assume("Jane","entity []","[] is human","[] is a child of []"
    ,"[] is a grandchild of []")
5 B=A.assume("[X][[X] is human]:[Y][[entity [Y]] is human][[X] is
     a child of [entity [Y]]]")
6 A.assume("[entity [Jane]] is human")
7 A.conclude(B, "entity [Jane]")
```

```
A.conclude(B,"entity [Y'']")
C=A.assume("[X][Y][Z][[X] is a child of [Y]][[Y] is a child of
    [Z]]:[[X] is a grandchild of [Z]]")
A.conclude(C, "entity [Jane]", "entity [Y'']", "entity [Y'''']")
D=A.conclude()
ada.note(D, "This is the concluding inference of the argument
    above")
```

```
[ADA] Notebook: Closing an argument

╔ Argument 0.0.
╠ Jane, entity *, * is human, * is a child of *, * is a grandchild
of * 0.1
╠ for X', given X' is human, conclusions Y', (entity Y') is
human, and X' is a child of (entity Y') hold 0.2
╠ (entity Jane) is human 0.3
║ Y", (entity Y") is human, (entity Jane) is a child of (entity
Y") 0.4
║ Y"', (entity Y"') is human, (entity Y") is a child of (entity
Y"') 0.5
╠ for X', Y', and Z', given X' is a child of Y' and Y' is a child
of Z', conclusion X' is a grandchild of Z' holds 0.6
║ (entity Jane) is a grandchild of (entity Y"') 0.7
╚ Y"', (entity Jane) is a grandchild of (entity Y"') 0.8

Expression 1:
[[[X'][Y'][Z'][[X'] is a child of [Y']][[Y'] is a child of
[Z']]:[[X'] is a grandchild of [Z']]][Jane][entity []][[] is
human]][[] is a child of []][[] is a grandchild of []][[X']][[X']
is human]:[Y'][[entity [Y']] is human]][[X'] is a child of
[entity [Y']]]][[entity [Jane]] is human]:[Y"'][[entity [Jane]]
is a grandchild of [entity [Y"']]]]
• This is the concluding inference of the argument above
```

Note that the purpose of the formulator `entity []` is to transform a formulator into a proposition. This is necessary in order to be able to suitably specialize the formulators in the premise of the inference, since a formulator cannot specialize to another formulator.

subarguments

An argument can be started within another argument, and its overall conclusion can be concluded within that other argument. Here is a variation of the argument from the previous section, which makes use of such *subargument*. #subargument

In the example below, we use the theme *theory* for the super argument. In this theme, assumptions are called *axioms* and conclusions are called *theorems*. Themes are inherited by subarguments, unless a subargument is assigned a different theme, which is the case below: its theme is *narrative*, which displays a deductive argument as a narrative. #theory #axioms #theorems #narrative

```
import ada
```

```
ada.notebook('Nested arguments')
A=ada.argument(theme="theory",name="general facts")
A.assume("entity []", "[] is human","[] is a child of []","[] is a grandchild of []", name="formulators")
B=A.assume("[X][[X] is human]:[Y][[entity [Y]] is human][[X] is a child of [entity [Y]]]", name="every human has a human parent")
C=A.assume("[X][Y][Z][[X] is a child of [Y]][[Y] is a child of [Z]]:[[X] is a grandchild of [Z]]", name="criteria for being a grandparent")
N=ada.argument(A, theme="narrative", name="a subargument")
N.assume("X","[entity [X]] is human")
N.conclude(B,"entity [X]")
N.conclude(B,"entity [Y'']")
N.conclude(C, "entity [X]", "entity [Y'']", "entity [Y''']")
N.conclude(["[[entity [Y''']] is human]","[[entity [X]] is a grandchild of [entity [Y''']]]"])
N.conclude()
A.conclude(N, name="every human has a human grandparent")
N.proof("fitch")
```

```
[ADA] Notebook: Nested arguments

Theory 0.0: general facts.

Notation 0.1: formulators.
[entity []][[] is human][[] is a child of []][[] is a grandchild
of []].

Axiom 0.2: every human has a human parent.
[[X'][[X'] is human]:[Y'][[entity [Y']] is human][[X'] is a
child of [entity [Y']]]].

Axiom 0.3: criteria for being a grandparent.
[[X'][Y'][Z'][[X'] is a child of [Y']][[Y'] is a child of
[Z']]:[[X'] is a grandchild of [Z']]].

Argument (0.4.0: general facts - a subargument). Let us suppose:
[X][[entity [X]] is human] (0.4.1). This gives us: [Y"][[entity
[Y"]] is human][[entity [X]] is a child of [entity [Y"]]]
(0.4.2). Hence: [Y"'][[entity [Y"']] is human][[entity [Y"]]
is a child of [entity [Y"']]] (0.4.3). This gives us: [[entity
[X]] is a grandchild of [entity [Y"']]] (0.4.4). So, [[entity
[Y"']] is human][[entity [X]] is a grandchild of [entity [Y"']]]
(0.4.5).

Argument complete. It proves that under the stated assumptions,
we have the following: [Y"'][[entity [Y"']] is human][[entity
[X]] is a grandchild of [entity [Y"']]] (0.4.6).

Theorem 0.5: every human has a human grandparent.
[[X'][[entity [X']] is human]:[Y""][[entity [Y""]] is
human][[entity [X']] is a grandchild of [entity [Y""]]]].
Proof.
╟ [X][[entity [X]] is human]
```

```
╟ [Y"][[entity [Y"]] is human][[entity [X]] is a child of [entity
[Y"]]]
╟ [Y"'][[entity [Y"']] is human][[entity [Y"]] is a child of
[entity [Y"']]]
╟ [[entity [X]] is a grandchild of [entity [Y"']]]
╟ [[entity [Y"']] is human][[entity [X]] is a grandchild of
[entity [Y"']]]
╙ QED
```

A subargument has access to all assumptions and conclusions in the super argument, but the converse is not the case. In particular, formulators pinned in the super argument remain pinned in the subargument, while formulators pinned in the subargument are not pinned in the super argument. In fact, the contents of the subargument has no effect whatsoever on the super argument, except that it does get stored in the super argument.

Note the *proof* of the theorem at the end. It is actually the same argument that was used to conclude the theorem, but displayed in the style of a mathematical proof, using the fitch theme. A proof only permits two themes: fitch and narrative. #proof

equivalence

The main application of deductive arguments is to establish *truth*. Here we are not talking merely about philosophical or even mathematical truth, but also truth as manifested in the world that surrounds us: things that are realisable, things that can be experienced or witnessed. The process of establishing such truth begins my *modelling* the phenomenon to which the truth will apply. This is done by formulating suitable assumptions (axioms). These assumptions constitute the *model*. They are meant to describe the logical structure of the phenomenon. Then, the conclusions that we can draw from the model by means of a repeated use of deductive arguments, will be establishing truth about the modelled phenomenon. Of course, the reliability of such truth is not absolute — it does depend on how accurate is the model. Experimentation gives us a certain degree of confidence about accuracy of the model, but such confidence is not a deductive one. Nevertheless, this mechanism is powerful enough to put a satellite on Earth's orbit, to send a wireless message from one place on Earth to another, to make an accurate X-ray image of a living organism, to build a computer, to warm up your food in the microwave, etc. In fact, all of the technological advances that humans have made rely, one way or another, on establishing truth using deductive arguments. #truth #modelling #model

One of the most important relations that arises in the applications of deductive reasoning is that of *equivalence*. Two propositions are equivalent when each of them can be inferred from the other. Below we illustrate how equivalence can be can be introduced and applied in [ADA]. #equivalence

```
1 import ada
2 ada.notebook(name="Modelling equivalence")
3 ada.note("- Abbreviations -")
4 _inf_=ada.note(ada.exp("[]:[]"), "Abbreviated as: _inf_")
5 X=ada.note(ada.exp("X"), "Abbreviated as: X")
```

```
Y=ada.note(ada.exp("Y"), "Abbreviated as: Y")
Z=ada.note(ada.exp("Z []"), "Abbreviated as: Z")
ada.note("- Pinned Formulators and Axioms -")
equivalence = ada.argument(theme="theory", name="equivalence",
    dialect="natural")
true=equivalence.assume("is true", name="proposition formulator
    ")
_equiv_=equivalence.assume("[] <=> []", name="equivalence
    formulator")
introEquiv=equivalence.assume(_inf_((X,Y,_inf_(X,Y),_inf_(Y,X))
    ,_equiv_(X,Y)), name="introduction of equivalence")
elimEquiv=equivalence.assume(_inf_((X,Y,_equiv_(X,Y)),(_inf_(X,
    Y),_inf_(Y,X))), name="elimination of equivalence")
ada.note("- Theorem -")
A=ada.argument(equivalence, name="Proving that equivalence is
    reflexive", theme="hidden")
A.assume(Z)
B=ada.argument(A)
B.assume(Z(true))
B.conclude([Z(true)])
B.conclude()
A.conclude(B)
A.conclude(introEquiv, Z(true), Z(true))
A.conclude()
theorem = equivalence.conclude(A)
A.proof()
ada.note("- Application -")
A=ada.argument(equivalence, name="sky is blue")
_isblue=A.assume("[] is blue")
sky=A.assume("sky")
A.conclude(theorem, _isblue(sky))
```

```
[ADA] Notebook: Modelling equivalence

- Abbreviations -

Expression 0:
[]:[]
• Abbreviated as: ᵢnf

Expression 1:
X
• Abbreviated as: X

Expression 2:
Y
• Abbreviated as: Y

Expression 3:
Z []
• Abbreviated as: Z

- Pinned Formulators and Axioms -

Theory 4.0: equivalence.

Notation 4.1: proposition formulator.
is true.
```

```
Notation 4.2: equivalence formulator.
* <=> *.

Axiom 4.3: introduction of equivalence.
for X' and Y', given for X', Y' holds and for Y', X' holds,
conclusion X' <=> Y' holds.

Axiom 4.4: elimination of equivalence.
for X' and Y', given X' <=> Y', conclusions for X', Y' holds and
for Y', X' holds hold.

- Theorem -

Theorem 4.6.
for yZ *, (yZ is true) <=> (yZ is true) holds.
Proof.
╠╚ Z *
╠╠╚ Z is true
╠╠╠ Z is true
╠╠ ( Z is true => Z is true )
╠╠ (Z is true) <=> (Z is true)
╚╚ QED

- Application -

Theory 4.7.0: equivalence - sky is blue.

Notation 4.7.1.
* is blue.

Notation 4.7.2.
sky.

Theorem 4.7.3.
(sky is blue) <=> (sky is blue).
```

Other ways of manipulating with propositions, such as the one used in propositional calculus or predicate calculus in mathematics, can be modelled in [ADA] in a similar fashion as we modelled equivalence above.

Activity

This activity is intended only for those who are familiar with axiomatic arithmetic in mathematics: an axiomatic theory of the natural number system. Model arithmetic in [ADA], following the approach due to Peano (which is based on Dedekind's definition of the natural number system). Formulate firstly all logical axioms needed for the development of arithmetic, and then prove that no natural number is a successor of itself. You might want to first establish a few intermediate logical theorems to simplify the required proof. [3]

[Chapter 5]
SOLUTIONS

Below you will find solutions to the activities throughout the book.

1

```
import ada
ada.notebook()
ada.save()
```

```
[ADA] Notebook:
```

2

```
import ada
```

```
ada.note("Here is what happens when you edit and save an [ADA] notebook without using the ada.notebook function:", "The heading disappears", "The name of the saved file is '.ada'")
ada.save()
```

Here is what happens when you edit and save an [ADA] notebook without using the ada.notebook function:

- The heading disappears
- The name of the saved file is '.ada'

3

```
import ada

ada.notebook("Arithmetic")

arithmetic = ada.argument(theme="theory", name="arithmetic", dialect="natural")

_inf_=ada.note(ada.exp("[]:[]"), "Abbreviated as: {\_}inf{\_}")
X=ada.exp("X")
Y=ada.exp("Y")
x=ada.exp("x")
y=ada.exp("y")
F=ada.exp("F []")
G=ada.exp("G []")

istrue = arithmetic.assume("is true", name="is true")
_equiv_ = arithmetic.assume("[] <=> []", name="equivalence")
_equal_ = arithmetic.assume("[] = []", name="equality")
false_ = arithmetic.assume("false []", name="fallacy")
_isfalse = arithmetic.assume("[] is false", name="negation")
num_ = arithmetic.assume("number []", name="number")
zero = arithmetic.assume("0", name="zero")
_isnat = arithmetic.assume("[] is a natural number", name="is a number")
_suc = arithmetic.assume("[]+1", name="successor")

introEquiv = arithmetic.assume(_inf_((X,Y,_inf_(X,Y),_inf_(Y,X)),_equiv_(X,Y)), name="introduction of equivalence")
elimEquivI = arithmetic.assume(_inf_((X,Y,_equiv_(X,Y)),_inf_(X,Y)), name="elimination of equivalence I")
elimEquivII = arithmetic.assume(_inf_((X,Y,_equiv_(X,Y)),_inf_(Y,X)), name="elimination of equivalence II")
defEq = arithmetic.assume(_inf_((X,Y),_equiv_(_equal_(X,Y),_inf_((F,G),_equiv_(F(X),F(Y))))), name="definition of equality")
defFallacy = arithmetic.assume(_equiv_(false_(istrue),_inf_(X,X)), name="definition of fallacy")
defNeg = arithmetic.assume(_inf_(X,_equiv_(_isfalse(X),_inf_(X,false_(istrue)))), name="definition of negation")
zeroNum = arithmetic.assume(_isnat(num_(zero)), name="zero is a natural number")
```

```
sucNum = arithmetic.assume(_inf_((x,_isnat(x)),(_isnat(_suc(x))
    )), name="successor of a natural number is a natural number"
    )
arithI = arithmetic.assume(_inf_(x,_isfalse(_equal_(num_(zero),
    _suc(x)))), name="zero is not a successor")
arithII = arithmetic.assume(_inf_((x,y),_inf_(_equal_(_suc(x),
    _suc(y)),_equal_(x,y))), name="successor is injective")
arithIII = arithmetic.assume(_inf_((F,F(num_(zero)),_inf_((x,
    _isnat(num_(x)),F(num_(x))),F(_suc(num_(x))))),_inf_((x,
    _isnat(x)),F(x))), name="induction principle")

E = ada.argument(arithmetic,theme="hidden", dialect="natural")
E.assume(F,G,F(istrue),_equiv_(F(istrue),G(istrue)))
EE = E.conclude(elimEquivI, F(istrue), G(istrue))
E.conclude(EE)
E.conclude()

elimEquivIII = arithmetic.conclude(E)
E.proof()

E = ada.argument(arithmetic,theme="hidden", dialect="natural")
E.assume(F,G,G(istrue),_equiv_(F(istrue),G(istrue)))
EE = E.conclude(elimEquivII, F(istrue), G(istrue))
E.conclude(EE)
E.conclude()

elimEquivIV = arithmetic.conclude(E)
E.proof()

E = ada.argument(arithmetic,theme="hidden", dialect="natural")
E.assume(F,F(istrue),_isfalse(F(istrue)))
E.conclude(defNeg, F(istrue))
EE=E.conclude(elimEquivIII, _isfalse(F(istrue)), _inf_(F(istrue
    ),false_(istrue)))
E.conclude(EE)
E.conclude()

introFalseI = arithmetic.conclude(E)
E.proof()

E = ada.argument(arithmetic,theme="hidden", dialect="natural")
E.assume(F,_inf_(F(istrue),false_(istrue)))
E.conclude(defNeg, F(istrue))
EE=E.conclude(elimEquivII, _isfalse(F(istrue)),_inf_(F(istrue),
    false_(istrue)))
E.conclude(EE)
E.conclude()

introIsFalse = arithmetic.conclude(E)
E.proof()

E = ada.argument(arithmetic,theme="hidden", dialect="natural")
E.assume(G, F,_inf_(G(istrue),F(istrue)),_isfalse(F(istrue)))
D = ada.argument(E)
D.assume(G(istrue))
D.conclude(_inf_(G(istrue),F(istrue)))
```

```
D.conclude(introFalseI, F(istrue))
D.conclude()
E.conclude(D)
E.conclude(introIsFalse, G(istrue))
E.conclude()

inherIsFalse = arithmetic.conclude(E)
E.proof()

E = ada.argument(arithmetic,theme="hidden", dialect="natural")
E.conclude(arithI, num_(zero))
A = ada.argument(E)
A.assume(x,_isnat(num_(x)))
A.assume(_isfalse(_equal_(num_(x),_suc(num_(x)))))
A.conclude(arithII, num_(x), _suc(num_(x)))
A.conclude(inherIsFalse, _equal_(_suc(num_(x)),_suc(_suc(num_(x
    )))), _equal_(num_(x),_suc(num_(x))))
A.conclude()
E.conclude(A)
E.conclude(arithIII, [_isfalse(_equal_("",_suc("")),[1,1]])
E.conclude()

arithmetic.conclude(E)
E.proof("fitch")
```

```
[ADA] Notebook: Arithmetic

Theory 0.0: arithmetic.

Expression 1:
[]:[]
• Abbreviated as: _inf_

Formulator 0.1: is true.
is true.

Formulator 0.2: equivalence.
* <=> *.

Formulator 0.3: equality.
* = *.

Formulator 0.4: fallacy.
false *.

Formulator 0.5: negation.
* is false.

Formulator 0.6: number.
number *.

Formulator 0.7: zero.
0.

Formulator 0.8: is a number.
* is a natural number.

Formulator 0.9: successor.
*+1.
```

```
Axiom 0.10: introduction of equivalence.
for X' and Y', given for X', Y' holds and for Y', X' holds,
conclusion X' <=> Y' holds.

Axiom 0.11: elimination of equivalence I.
for X' and Y', given X' <=> Y', conclusion for X', Y' holds holds.

Axiom 0.12: elimination of equivalence II.
for X' and Y', given X' <=> Y', conclusion for Y', X' holds holds.

Axiom 0.13: definition of equality.
for X' and Y', (X' = Y') <=> (for BF * and DG *, (BF X') <=> (BF
Y') holds) holds.

Axiom 0.14: definition of fallacy.
(false is true) <=> (for X', X' holds).

Axiom 0.15: definition of negation.
for X', (X' is false) <=> (for X', false is true holds) holds.

Axiom 0.16: zero is a natural number.
(number 0) is a natural number.

Axiom 0.17: successor of a natural number is a natural number.
for x', given x' is a natural number, conclusion x'+1 is a
natural number holds.

Axiom 0.18: zero is not a successor.
for x', ((number 0) = x'+1) is false holds.

Axiom 0.19: successor is injective.
for x' and y', ( x'+1 = y'+1 => x' = y' ) holds.

Axiom 0.20: induction principle.
for sF *, given sF number 0 and for x', given (number x') is
a natural number and sF number x', conclusion sF (number
x')+1 holds, conclusion for x', given x' is a natural number,
conclusion sF x' holds holds.

Theorem 0.22.
for XF * and gG *, given XF is true and (XF is true) <=> (gG is
true), conclusion gG is true holds.
Proof.
╠ F *, G *, F is true, (F is true) <=> (G is true)
║ ( F is true => G is true )
║ G is true
╚ QED

Theorem 0.24.
for yF * and yG *, given yG is true and (yF is true) <=> (yG is
true), conclusion yF is true holds.
Proof.
╠ F *, G *, G is true, (F is true) <=> (G is true)
║ ( G is true => F is true )
║ F is true
╚ QED
```

```
Theorem 0.26.
for nF *, given nF is true and (nF is true) is false, conclusion
false is true holds.
Proof.
╠═╗ F *, F is true, (F is true) is false
╠ ╗ ((F is true) is false) <=> (( F is true => false is true ))
╠ ╗ ( F is true => false is true )
╠ ╗ false is true
╚ ╝ QED

Theorem 0.28.
for jF *, given ( jF is true => false is true ), conclusion (jF is
true) is false holds.
Proof.
╠═╗ F *, ( F is true => false is true )
╠ ╗ ((F is true) is false) <=> (( F is true => false is true ))
╠ ╗ ( ( F is true => false is true ) => (F is true) is false )
╠ ╗ (F is true) is false
╚ ╝ QED

Theorem 0.30.
for xG * and PF *, given ( xG is true => PF is true ) and (PF is
true) is false, conclusion (xG is true) is false holds.
Proof.
╠═╗ G *, F *, ( G is true => F is true ), (F is true) is false
╠ ╠═╗ G is true
╠ ╠ ╗ F is true
╠ ╠ ╗ false is true
╠ ╗ ( G is true => false is true )
╠ ╗ (G is true) is false
╚ ╝ QED

Theorem 0.32.
for x", given x" is a natural number, conclusion (x" = x"+1) is
false holds.
Proof.
╠ ╗ ((number 0) = (number 0)+1) is false
╠ ╠═╗ x', (number x') is a natural number
╠ ╠═╗ ((number x') = (number x')+1) is false
╠ ╠ ╗ ( (number x')+1 = ((number x')+1)+1 => (number x') = (number
x')+1 )
╠ ╠ ╗ ((number x')+1 = ((number x')+1)+1) is false
╠ ╗ for x', given (number x') is a natural number and ((number x')
= (number x')+1) is false, conclusion ((number x')+1 = ((number
x')+1)+1) is false holds
╠ ╗ for x', given x' is a natural number, conclusion (x' = x'+1) is
false holds
╚ ╝ QED
```

Index

[ADA]

an assembler of deductive arguments

by Zurab Janelidze

ADA presents an innovative blend of formal mathematical language and Python programming, designed to teach deductive argument construction in an intuitive way. Aimed at readers ranging from programming enthusiasts to students and professionals in linguistics, mathematics, logic, and philosophy, the book offers a narrative approach to the art of deductive reasoning, making it appealing to lovers of literature and linguistics. It serves as a novel introduction to the discipline, guiding Python-savvy individuals through the intricacies of deductive arguments and the elegance of mathematical thought. This multifaceted work is as much a tutorial in formal reasoning as it is a fresh perspective on mathematical languages, promising to engage and enlighten a broad spectrum of curious minds.