Literate programming version of the graph stratification algorithm version of the Marcel prover

Lavinia Randall Holmes

June 7, 2026

Contents

0.1 version remarks 2

1 Introduction **3**
1.1 Philosophical considerations 5

2 The code **7**
2.1 The term and formula types 7
2.2 Display functions and parser 11
2.3 Substitution functions 20
2.4 The graph based stratification algorithm 28
2.5 The definition mechanism 37
2.6 The sequent prover (with complete user command list for the software as a whole) 45
2.7 A note on equality 58
2.8 The rules 65
2.8.1 Specific Sequent Rules: Connectives 65
2.8.2 Axiom 65
2.8.3 Left rules (Premise rules) 65
2.8.4 Right rules (Conclusion rules) 66
2.8.5 More Sequent Rules 66
2.8.6 Rules for Quantifiers 66
2.8.7 Left Rules (Premise Rules) 66
2.8.8 Right Rules (Conclusion Rules) 66
2.8.9 Comments on Quantifier Rules 67
2.8.10 Rules for Membership 67
2.8.11 Left Rule 67
2.8.12 Right Rule 67
2.8.13 Rules for Equality 67
2.8.14 Left Rule 67

2.8.15 Right Rule 67
2.8.16 Global Substitution 68

3 Some sample script files **78**
3.1 Inclusion is transitive 78
3.2 Equality is transitive 96
3.3 Coextensionality implies Leibniz equality 122
3.4 A weird example with unknown variables 149
3.5 Atomic inclusion is membership 156
3.6 Projection theorems for the ordered pair 169
3.7 Foundations of arithmetic 406

0.1 version remarks

Graph stratification version of the Marcel theorem prover, 6/7/2026 by (Lavinia) Randall Holmes, intellectual property rights to be respected to the extent of preserving this attribution, please.

6/7/2026: savethproof and loadproof commands now are recorded to logs

6/6/2026: The `savetheproof` and `loadproof` arguments have been installed, allowing one to leave a proof, prove a required lemma, then rejoin it at the same point.

6/2/2026: Installed digits as definienda. Fixed harmless bug which posted definienda to the variable list. Installed a brief construction for entering arguments to let terms.

5/31/2026: Notes to self as I edit. The occurrence index counter might need to be updated when `theoremcut` is executed. This might be faked using a substitution. Addressed this issue: the cut instance of the theorem has all new occurrence indices. It is worth noting (because this concerned me) that there is no such problem for definitions: defexpand already updates occurrences.

It remains interesting to try to improve the display of stratifications, though the ones actually displayed in normal work, the cycles witnessing stratification failure and the type tables for definitions, are actually pretty readable at this point.

5/29/2026: The self-documented literate programming version of this code.

5/28/2026: The literate programming version. Getting rid of most earlier update comments.

5/27/2026: Quite a lot has been added today.

1. Made the variable space used for let terms (definitions) completely disjoint from variables which can be entered or displayed. This makes evaluation of definitions secure from variable capture problems.

2. Fixed soundness problems with dynamic setting of unknowns. An unknown is to be matched only once, and its match is recorded (for the equality function to make checks for consistency). Also, realized that higher order matching had no unknown dependency checks.

3. Further, equality tests will now attempt to establish equality by definition expansion. This may have various fancy effects.

It is a fun point that though I did a lot of work today it appears to have no effect on existing scripts.

5/27/2026: The equality functions now test whether definition expansion will make terms equal.

The unknown setting feature is unsound and needs to be repaired. installed a fix: dare I test it? Fix seems to work...

It is very important that done() appears to be the only client of the equality tests. Any client needs to appropriately reset unknownassignments.

Also added unknown dependency checks to the higher order matching. There needs to be a lot of tightening.

5/27/2026: More variable security needed for the definition facility. the @ variables used in substitutions into definitions cannot be allowed to be generated in user entered or even generated text.

The appearance is exactly the same as before, but function names in the code may be annoying due to the history of my thought process while correcting this problem : the hash is now the variable prefix used internally (so atify functions add hashes) and the at sign @ is used externally as before (so hashify adds @ and deatify changes to @) But the variables used for definition (let term) computation are now completely secure: no user operation can produce them, so no variable capture can occur in let term computation.

theoremcut now atifies ("hashifies") the new formula introduced, so that free variables in theorems will not intrude in the usual namespace. they can be manipulated using setunknown to be replaced with terms containing no fresh variables, which gives them very limited privileges. Free variables left over from definitions will be treated similarly.

I still havent fixed apparent variable capture ;-) It is philosophically appealing but probably not practical that I somehow do not have to.

1 Introduction

This is an alternative development of the core of the Marcel sequent prover for stratified set theory. The immediate occasion for starting this project was consideration of Ryan Nolan's use of the Bellman-Ford algorithm to test stratification: using Bellman-Ford is a bad idea, because the second time that you

relax or even have the opportunity to extend an edge [something that Bellman-Ford won't even notice] you should abort and report stratification failure. Here I describe a more sensible graph based algorithm for stratification testing, which can be thought of as a variant of Dijsktra's algorithm. This is also informed by the fact that Forster and I are working right now, quite independently, on the properties of the graph determined by the atomic formulas in a formula: we are particularly interested in connectedness of this graph, and the stratification testing in this version is very weak, even weaker than Marcel's weak stratification: all that is required is that occurrences of variables connected to the binding variable in a set abstract be typed sensibly relative to that variable. Variables in terms and formulas of the language used here have occurrence data attached to them invisibly [and it is occurrences that have relative type, not variables per se], which has odd effects, not only enabling weak stratification, but also removing the effects of bound variable capture (though I really should modify the display function to warn the viewer about apparent bound variable capture: the system can tell when some variables in the scope of a bound variable are actually free with the same shape, but it does not warn the viewer when this happens; the system does prevent the binding of the free "fresh variables" introduced by the quantifier rules.

It is important to note that the basic stratification condition, that the lengths of all paths between the same pair of variables are the same, was stated (in an incorrect form) by Quine in the original NF paper, and the correction allowing paths to contain converse membership (with negative length) as well as membership with positive length (and equality with zero length) was proposed (as a correction to Quine) by Thomas Forster years ago. Nolan was an incidental inspiration to the present effort, but had not proposed anything not prefigured in earlier work: and what we do is actually quite different. It is a nod to Nolan that the data reported on stratification failure is a cycle of nonzero length in the graph associated with a formula.

This version has NF extensionality, but could easily be converted to SF or NFU. I do think there are interesting advantages to working with a fully extensional theory.

Other things to notice is that this version has Polish notation input (so far): this is handy for quick prototyping. The output notation is somewhat more familiar.

The sequent prover is quite similar to previous versions, but if anything more streamlined. Marcel enjoys a very limited set of commands, as the execution of most sequent rules is simply "apply the appropriate rule to the left or right sequent". Using a list with reference pointers gives a much more sensible proof structure than the tree structure I used in previous versions. The very efficient and liberal stratification algorithm is nice, though I think the method I used last time of requiring type indices on bound variables worked quite well as well.

The way I handled defined notions is new with this version. It is motivated in some details by the needs of the graph stratification algorithm. It allows introduction of defined notions with any arity and allows arguments to be supplied in any order (though I haven't exploited this). The support for infix display is

amusing.

The automatic matching of unknown variables is expanded to higher order in this version. The idea is to avoid having to use `setunknown` too often. The fact that my scripts are actually Python programs may be useful, and the addition of display information to them makes them more intelligible and perhaps easier to debug, though larger.

Things which could be added:

- Decluttering of sequents (pruning propositions from left or right) should be supported. The autopruning feature of previous versions of Marcel makes for more economical saved formal proofs.
- It would be nice to have a parser for something more like the output notation. An approach might be to parse with the Polish notation as the target, then check by applying the display function that we got the right thing.
- Automatic generation of cut free proofs (normalization) would be interesting.
- I have never made any particular attempt to write tactics for Marcel in any of its versions. Perhaps I should think about this.
- I should work on an interactive script editor to make it easier to correct a script (if the proof strategy needs to be changed, or if a change in the code messes up its behavior). One envisions the ability to run a script in demo mode, but also to be able to drop in and change a command and/or insert or delete some commands, and then resume. A related idea would be construction of proofs from scripts, or designing scripts to be as far as possible stable after code changes.
- Can the definition facility be modified to enable safe use of definitions of unstratified concepts?
- Do we want additional primitives, such as pairs and projections, relation and function abstraction, and so forth? What about defined binding constructions?

1.1 Philosophical considerations

The author is tempted to contemplate Marcel as an implementation of the program of the original New Foundations paper. We maintain, in line with what Quine actually says, is that the purpose of the paper was not to propose a weird new set theory, though that was its practical effect. The purpose of the paper was to explore the logicist viewpoint of the foundations of mathematics.

Quine was at pains to minimize his primitives (probably to be sure that they were logical). He uses the Sheffer stroke and the universal quantifier to cover the ground of first order logic. We adopt more primitives, but of course they

are all definable in terms of Quine's primitives, and Quine defines all of them and makes essential use of them. It might be fun in this connection to add the Sheffer stroke with suitable sequent rules to our machinery, to support the demonstration of its sufficiency. Our style of first order logic proof is different from Quine's. But the state of the art has improved.

The final primitive is membership. In this connection we have two concerns to air.

The first is whether membership is a logical notion. What we say on this is, *predication* is certainly a logical notion, and membership is a case of predication. The additional element is that we are given a criterion (stratification) for allowing reification of a predicate. Reification of predicates is needed because in $x \in y$, y represents a predicate to be considering as holding or not holding of x, but it is also an object. Some predicates are being taken to be objects. Some cannot be so taken, the index example being non-self-membership. Quine's adoption of this criterion was purely pragmatic. We believe that there may be a way to motivate it which preserves the impression that it might be a logical notion. We need a reason to believe that the stratification criterion picks out those predicates which are "genuine" in some sense. We have elsewhere suggested such a motivation, based on the idea that stratified predicates are those which are invariant in a suitable sense under redefinitions of the membership relation by permutation...the underlying idea being that the identification of predicates with particular objects required to implement membership is extrinsic to the predicates and to the objects, and that genuine properties of objects should be invariant under redefinition of membership. There is also the original view which probably accurately represents Quine's thinking: logic of order ω is logic. Then the collapse of the types into one can be evaluated as a pragmatic move to simplify matters, still leaving us with an implementation of logic of order ω, though incidentally creating more predicates which we cannot (or at least do not) reify.

The second is whether we should at the same time introduce other related notions, such as abstract relations or functions (for which pairs are useful). In the domain of first order logic we did this, but Quine's working set of concepts includes all of the ones we used. We have not so far chosen to implement relational or functional abstraction directly in this system [though we did implement lambda terms, and also pairs and their projections, as primitives in previous versions of Marcel]. So far we are choosing to keep membership and set abstraction as our primitives. Quine did not take set abstraction as primitive: he defined set abstraction as a species of definite description, and used a contextual definition of definite description, following Russell. We note that we can define (stratified) definite description using set abstraction. What we *have* done is provided a systematic way to introduce defined notions (with both term and formula values); we aim for a reasonably fluent treatment of pairs, relations and functions without introducing new primitive constructions. It is an adventure.

We are letting the logic of this prover be New Foundations. In earlier versions of Marcel, we implemented NFU; the original motivation was to implement

Marcel Crabbé's sequent calculus SF for stratified set theory with no extensionality at all but with a set abstraction construction, for which he was able to prove cut elimination. New Foundations is very convenient in certain ways. We do want to bear in mind how the logic could be weakened to NFU, and how the development of a body of mathematics would change.

The mathematical project we have in mind is to carry out Specker's proof of the "Axiom" of Infinity in NF. This will involve developing arithmetic, which is always interesting.

2 The code

The initial utilities handle things to do with scripting. Marcel generates a Python script which will repeat a session, if prompted correctly.

```
  ’’’

logfilename="testlog,txt"

logfile=open(logfilename,’a’)

# USER COMMAND

# set the log file to be used.

def setlog(s):
    global logfilename
    global logfile
    logfile.close()
    logfilename="logs/"+s+’logfile.py’
    logfile=open(logfilename,’w’)
    logfile.write(’from graph2 import *\n\n’)

def usercommand(s):

    if demostate==True: print(s)

    logfile.write(s)

r’’’
```

2.1 The term and formula types

The primitive relations of the implemented theory are equality `=` and membership `e`.

The logical connectives are the usual conjunction (`&`), disjunction (`V`), implication (`>`) and biconditional (`X`). Implication and biconditional are displayed in

the more familiar looking forms `->` and `==`. Negation ($\sim$) is a basic construction in the formula type. The quantifiers are the usual universal (`A`) and existential (`E`) quantifiers.

There is a construction for defined predicates to be discussed later.

Terms are either variables (single lower case letters possibly followed by one or more primes ‘ or `*`), set abstracts, or defined terms, to be discussed later.

There is a much more complete discussion of the language of Marcel below.

```
'''

relations = ["e","="]

def isrelation(s):
    return s in relations

connectives = ["&","V",">","X"]

# < is implication and X is biconditional

def isconnective(s):
    return s in connectives

# this function changes the shape of some connectives in the display.

def cexp(s):
    if s==">":  return "->"
    if s=="X":  return "=="
    return s

quantifiers = ["A","E"]

def isquantifier(s):
    return s in quantifiers

# we could add more primes to diversify the variables.
# Actually, digits could play this role.

primes = ["'","*"]

def isprime(s):
    return s in primes

# other components we might want would be pairs, lambda terms.

def isformula(L):
    if len(L)==0:  return False
```

```
    #atomic formula
    if isrelation(L[0]) and isterm(L[1]) and isterm(L[2]):
        return True
    #negation
    if L[0] == "~" and isformula(L[1]):return True
    #binary propositional connective
    if (isconnective(L[0])
        and isformula(L[1])
        and isformula(L[2])):  return True
    #quantified formula
    if (isquantifier(L[0])
        and type(L[1]) == str
        and type(L[2]) == int
        and isformula(L[3])):  return True
    # formula let terms look typgraphically the same as term let terms:
    # what this does is force formula definitions to have arguments)
    # we allow defined constant terms but not defined constant formulas
    # let term
    if L[0] == "let" and isterm(L):  return True

    return False

# isterm does not do a stratification check.  Does it need to?

def isterm(L):
    if len(L)==0:  return False
    #variable
    if (L[0] == "var"
        and type(L[1]) == str
        and type(L[2]) == int):  return True
    #set abstract
    if (L[0] == "set"
        and type(L[1]) == str
        and type(L[2]) == int
        and isformula(L[3])):  return True
    #defined
    if L[0] == "defined" and type(L[1]) == str: return True
    #let term
    if (L[0] == "let"
        and isterm(L[1])
        and L[1][0]=="var"
        and isterm(L[2])
        and isterm(L[3])
        and (L[3][0]=="defined"
             or L[3][0]=="let")):  return True
    return False
```

```
r'''
```

An important but invisible feature of term and formula management in the prover is occurrence indexing of variables. Each occurrence of a variable has a different index, except that variables bound by the same quantifier or set abstract are identified as the same occurrence. It is occurrences, not variables per se, that are assigned relative type in the stratification algorithm. Free occurrences of a variable, though they are all differently indexed as occurrences, in fact represent the same object. Occurrence indices are not displayed. The functions below are used to renumber occurrences indices of variables bound by the same binder so that they are the same: the intended condition is that this is the *only* situation in which occurrence indices of variables are the same.

```
'''
# these functions are used to renumber occurrences
# of variables which are bound
# by the same quantifier or abstract:
# such occurrences are identified.

def reinstt(s,n,L):
    if (L[0]=="var"
        and L[1] == s
        and type(L[2])==int):
        return [L[0],L[1],n]
    if (L[0]=="var"
        and type(L[1])==str
        and type (L[2])==int):return L
    if (L[0]=="set"
        and L[1] == s
        and type(L[2])==int
        and isformula(L[3])): return L
    if (L[0]=="set"
        and type(L[1])==str
        and type (L[2])==int
        and isformula (L[3])):
        return [L[0],L[1],L[2],
                reinstf(s,n,L[3])]
    if L[0]=="defined": return L
    if L[0]=="let":
        return [L[0],L[1],
                reinstt(s,n,L[2]),
                reinstt(s,n,L[3])]
    return "!!!"

def reinstf(s,n,L):
```

```
    if (isrelation(L[0])
        and isterm(L[1])
        and isterm(L[2])):
        return [L[0],reinstt(s,n,L[1])
                ,reinstt(s,n,L[2])]
    if L[0]=="~" and isformula(L[1]):
        return [L[0],reinstf(s,n,L[1])]
    if (isconnective(L[0])
        and isformula(L[1])
        and isformula(L[2])):
        return [L[0],reinstf(s,n,L[1])
                ,reinstf(s,n,L[2])]
    if (isquantifier(L[0])
        and L[1]==s
        and type(L[2])== int
        and isformula(L[3])):  return L
    if (isquantifier(L[0])
        and type(L[1])==str
        and type(L[2])==int
        and isformula(L[3])):
        return [L[0],L[1],L[2],
                reinstf(s,n,L[3])]
    if L[0]=="let":
        return [L[0],L[1],
                reinstt(s,n,L[2]),
                reinstt(s,n,L[3])]
    # term function in the formula
    # third argument is not a mistake
    return "???"

r'''
```

2.2 Display functions and parser

For ease of prototyping, the display notation and the input notation of this prover are (at least up to this point) quite different. Terms and formulas are input in Polish notation and output in a rather more familiar looking style.

We describe the language of the prover, in standard mathematical terms and in terms of input and output.

Formulas are as follows

- Atomic formulas are identity statements and membership statements. If t and u are terms, the mathematical notations are $t = u$, $t \in u$ for identity and membership statements respectively. The input notations are `=tu`, `etu` (where t and u are expressed in input notation). The output notations are `t = u` and `t e u` respectively (where t and u are expressed in output

notation).

- The binary propositional connectives are conjunction, disjunction, implication and biconditional. Where p and q are formulas, $(p \wedge q)$, $(p \vee q)$, $(p \rightarrow q)$ and $(p \leftrightarrow q)$ are the respective mathematical notations, implemented as input by `&pq`, `Vpq`, `>pq`, `Xpq` and as output by `(p & q)`, `(p V q)`, `(p -> q)`, `(p == q)`.

- Negation is a separate construction because of arity. Where p is a formula, $\neg p$ is its negation, with input and output notation $\sim$`p`.

- Quantified formulas are of the usual forms $(\forall x.\phi)$, $(\exists x.\phi)$, for which input notation is `Ax`ϕ and `Ex`ϕ and output notation is `(Ax :` ϕ`)` and `(Ex :` ϕ`)` where `x` is a variable and ϕ is a formula.

- A notation abbreviating a formula or term is either a capital letter X or of one of the forms K' or K^* where K is a notation abbreviating a term or formula. The input notation is `X`, `‘K`, `*K` and the output notation is `X`, `K’`. `K*`. The same notation may serve as a term defined notion and a formula defined notion independently.

  A defined formula is either of the form $K[t/a]$ or $\Delta[t/a]$, where K is a notation abbreviating a formula, a is a variable (normally appearing in K), t is a term, and $K[t/a]$ represents the result of replacing a with t in the formula abbreviated by K, and $\Delta[t/a]$ represents the result of replacing a with t in the expanded version of the defined formula Δ. The namespaces of variables used in defined notions K and in the ordinary language of the prover are disjoint though the surface notations for the variables are the same (we will see how this is implemented). The input notation for this is `:at`K, resp. `:at`Δ. The output notation is `K(a:t)`, resp. Δ`(a:t)`. A convenient exception in the output notation is that `:at:bu`Δ expands as `(t` Δ `u)`, where the keys are literally `a` and `b`.

Terms are as follows:

- Variables are lower case letters, possibly differentiated with primes. A variable is either `x` (any lower case letter) or v' or v^* where v is a variable. The input notation is `x`, `‘v`, `*v` [sorry, but this lets the Polish notation work cleanly] and the output notation is the more sensible `x`, `v’`, `v*`.

  Fresh variables introduced by the quantifier rules are suffixed with `!` (arbitrary variables) or `?` (unknown variables, which can later be set to specific values). These suffixes are not understood by the parser. The parser will not allow a variable which has been used as a fresh variable to be bound.

  The user should be warned that there are contexts in user commands where variables are entered in output notation (without any `!` or `?`). There are additional variables `@v` which cannot be input by the user (generated by expansion of definitions or use of theorems): they can be addressed with some commands using their output notation. There are internal variables

`#v` used in the innards of the definition feature which should never be seen by the user.

Variables have occurrence indices which are managed by the prover and never seen by the user. Two occurrences of a variable have the same occurrence index precisely if they are bound by the same binder (quantifier or set abstraction).

- Set abstracts are of the form $\{v \mid \phi\}$ where ϕ is a formula and v is a variable. The input notation is {vϕ and the output notation is {`v | `ϕ}. It is useful to note that the parser will not accept set abstracts which are not stratified [in a weak sense to be described].

- A defined term is either of the form K or $\Delta[t/a]$, where K is a notation abbreviating a term [defined above in the clause on defined formulas], a is a variable (normally appearing in the term represented by K), t is a term, and $\Delta[t/a]$ represents the result of replacing a with t in the expanded version of the defined term Δ. The namespaces of variables used in defined notions K and in the ordinary language of the prover are disjoint (we will see how this is implemented). The input notation for this is `K`, resp. `:at`Δ. The output notation is `K`, resp. Δ`(a:t)`. A convenient exception in the output notation is that `:at:bu`Δ expands as `(t `Δ` u)`, where the keys are literally `a` and `b`.

  Note that all defined formulas are typographically identical to defined terms, a fact which the prover takes advantage of in various places. The same notation can be used to denote a defined term and an (obviously different) defined formula. Use of notations without substitutions for formulas is excluded because a need for these is not seen, and there are other uses of capital letters initial to input notations for formulas.

A new feature: the form `.tu` abbreviates `:atu` while `.t.uv` abbreviates `:at:buv`, `.t.u.vw` abbreviates `:at:bu:cvw` and so forth. This works for both terms and formulas.

```
,,,

# display functions

# the display of terms highlights things
#which might be "fresh variables"
# introduced by the quantifier rules,
# though it doesn't test whether they are free
# (the parser has now been changed so that
# fresh variables cannot be bound, even apparently:
# no bound variable can have the same surface form).

# added infix display of let terms (for both terms and formulas)
```

```
# if the first two keys used are a and b

# function to insert line breaks
# to avoid margin overflow.  The criterion is
# a bit weird.

def maybebreak(L,M):
    if ((len(str(L))>25 and len(str(M))> 60)
        or (len(str(M))>25 and len(str(L))> 60)):
        return "\n   "
    return " "

def displayf(L):
    global unknowns
    global freshvars
    if (isrelation(L[0])
        and isterm(L[1]) and isterm(L[2])):
        return ((displayt(L[1]))
                +(maybebreak(L[1],L[2]))
                +L[0]+" "+(displayt(L[2])))
    if L[0] == "~" and isformula(L[1]):
        return "~" + (displayf(L[1]))
    if (isconnective(L[0])
        and isformula(L[1]) and isformula(L[2])):
        return ("("+(displayf(L[1]))
                +(maybebreak(L[1],L[2]))
                +cexp(L[0])+" "+(displayf(L[2]))+")")
    if (isquantifier(L[0])
        and type(L[1])==str
        and type(L[2]) == int
        and isformula(L[3])):
        return ("("+L[0]+L[1]+" : "
                +(displayf(L[3]))+")")
    if L[0]=="defined" and type(L[1]) == str: return L[1]
    if (L[0]=="let"
        and isterm(L[1])
        and L[1][0]=="var"
        and L[1][1]=="a"
        and isterm(L[2])
        and L[3][0]=="let"
        and isterm(L[3][1])
        and L[3][1][0]=="var"
        and L[3][1][1]=="b"
        and isterm(L[3][2])
        and isterm(L[3][3])):
        return ("("+(displayt(L[2]))
```

```
                +" "+(displayf(L[3][3]))
                +" "+(displayt(L[3][2]))+")")

    if (L[0]=="let"
        and isterm(L[1])
        and L[1][0]=="var"
        and isterm(L[2])
        and isterm(L[3])
        and (L[3][0]=="defined"
             or L[3][0]=="let")):
        return (displayt(L[3])+"("+displayt(L[1])
                +":"+displayt(L[2])+")")
    return "???"

def displayt(L):
    global unknowns
    global freshvars
    if (L[0]=="var"
        and not(findunknown(L[1])=="error")):
        return L[1]+"?"
    if L[0]=="var" and L[1] in freshvars:
        return L[1]+"!"
    if (L[0]=="var"
        and type(L[1]) == str
        and type(L[2]) == int):
        return L[1]
    if L[0]=="defined" and type(L[1]) == str:
        return L[1]
    if (L[0]=="let" and isterm(L[1])
        and L[1][0]=="var" and L[1][1]=="a"
        and isterm(L[2]) and L[3][0]=="let"
        and isterm(L[3][1]) and L[3][1][0]=="var"
        and L[3][1][1]=="b" and isterm(L[3][2])
        and isterm(L[3][3])):
        return ("("+(displayt(L[2]))
                +(maybebreak(L[2],L[3][2]))
                +(displayt(L[3][3]))
                +" "+(displayt(L[3][2]))+")")
    if (L[0]=="let" and isterm(L[1])
        and L[1][0]=="var" and isterm(L[2])
        and isterm(L[3]) and (L[3][0]=="defined"
                              or L[3][0]=="let")):
        return (displayt(L[3])+"("+displayt(L[1])
                +":"+displayt(L[2])+")")
    if (L[0]=="set"
        and type(L[1]) == str
```

```
        and type(L[2]) == int
        and isformula(L[3])):
        return "{"+L[1]+" | "+(displayf(L[3]))+"}"
    return "!!!"

# parser

newint = 1

def shownewint():

    print(newint)

# note that the parser enforces
#stratification of set abstract terms.
# However, it does this in quite a weak sense.
# Types are assigned
# to occurrences of variables,
# not to variables themselves;  the only
# occurrences which are identified
# are those bound by the same abstract
# or quantifier.  Further, the
#stratification test for abstracts only
# checks variables which are connected
# to the variable bound in the abstract.
# this is all that is necessary;
# it is even less than Marcel's weak
# stratification criterion.

# the list of all variables used

variables=[]

def showvariables():
    print(variables)


# the list of all "unknowns"
# (witnesses introduced for later specification)

unknowns=[]

def showunknowns():
    print(unknowns)

# the list of all fresh variables
```

```
# (arbitraries and unknowns) introduced by quantifier rules

freshvars=[]

def showfreshvars():
    print(freshvars)

# The parser:  for rapid prototyping,
# this is Polish notation.
# The display notation is more standard.
# Examples when they are produced
# will give an indication of
# how to enter terms and formulas.

# Weak stratification checking
# of set abstracts is done at parse time.
# The graph generation
# and stratification functions are below.

def isspace(s):
    return (s==" " or s=="("
                          or s == ")" or s == "["
                          or s == "]" or s ==",")
# these allow organization of the Polish notation input

def applyprime(p,L):
    if len(L)==0: return "!!!"
    return [L[0]]+[L[1]+p]+L[2:]
# this makes it possible to prime both variables and definienda.

def raisekey(t):
    if t[0]=="let" and len(t[1][1])==1:
        return ["let",gett(chr(ord(t[1][1])+1)),t[2],raisekey(t[3])]
    return t

def gett(s):
    global newint
    global variables
    if len(s)==0: return "!!!"
    if isspace(s[0]): return gett(s[1:])
    if isprime(s[0]):
        v=applyprime(s[0],gett(s[1:]))
        if v == "!!!":  return "!!!"
        if (("a" <= v[1][0] and v[1][0] <= "z")
            and not(v[1] in variables)):
            variables=variables+[v[1]]
```

```
        return v
    if "a" <= s[0] and s[0] <= "z":
        if not(s[0] in variables):
            variables=variables+[s[0]]
        newint=newint+1
        return ["var",s[0],newint]
    if (("A" <= s[0] and s[0] <= "Z")
        or ("0" <= s[0] and s[0] <= "9")):
        return ["defined",s[0]]
    if s[0]==":":
        V=gett(s[1:])
        T=gett(restt(s[1:]))
        B=gett(restt(restt(s[1:])))
        if not(V[0]=="var"):
            return "!!!"
        if (not(B[0]=="defined"
                or B[0]=="let")):
            return "!!!"
        return ["let",V,T,B]
    if s[0]==".":
        T=gett(s[1:])
        B=gett(restt(s[1:]))
        if (not(B[0]=="defined"
                or B[0]=="let")):
            return "!!!"
        return ["let",gett("a"),T,raisekey(B)]
    if s[0]=="{":
        var=gett(s[1:])

        if not (var[0]=="var"):  return "!!!"

        if var[1] in freshvars:
            print("Attempt to bind a fresh variable")
            return "!!!"
        FF=   getf(restt(s[1:]))
        if not(isformula(FF)):  return "!!!"
        F=reinstf(var[1],var[2],FF)
        S=stratify(graphf(F),[var[1],var[2]])
        if S[0]=="stratification failure":
            print(displayf(F))
            print(S)
            return "bad set abstract"
        return ["set",var[1],var[2],F]
    return "!!!"

def restt(s):
```

```
    if len(s)==0:  return ""
    if isspace(s[0]):  return restt(s[1:])
    #if s[0]=="#":  return restt(s[1:])
    if isprime(s[0]):  return restt(s[1:])
    if "a"<=s[0] and s[0]<="z": return s[1:]
    if (("A"<=s[0] and s[0]<="Z")
        or ("0" <= s[0] and s[0] <= "9")): return s[1:]
    if s[0]==":":  return restt(restt(restt(s[1:])))
    if s[0]==".":  return restt(restt(s[1:]))
    if s[0]=="{":
        var=gett(s[1:])
        if not(var[0]=="var"):  return ""
        return restf(restt(s[1:]))
    return ""



def getf(s):
    global newint
    global variables
    if len(s)==0:  return "???"
    if isspace(s[0]):  return getf(s[1:])
    if len(s)==0:return "???"
    if isrelation(s[0]):
        return [s[0],gett(s[1:]),gett(restt(s[1:]))]
    if s[0]=="~":return [s[0],getf(s[1:])]
    if isconnective(s[0]):
        return [s[0],getf(s[1:]),getf(restf(s[1:]))]
    if isquantifier(s[0]):
        var=gett(s[1:])
        if not (var[0]=="var"):  return "???"
        return [s[0],var[1],var[2],
                reinstf(var[1],var[2],
                        getf(restt(s[1:])))]
    if s[0]==":" or s[0]==".": return gett(s)
    # getting a term here is a trick;
    # let terms have same form in both classes, and in this way
    # we forbid defined formulas without assignments to arguments.
    return "???"

def restf(s):
    if len(s)==0:return "???"
    if isspace(s[0]):return restf(s[1:])
    if isrelation(s[0]): return restt(restt(s[1:]))
    if s[0]=="~":return restf(s[1:])
    if isconnective(s[0]):return restf(restf(s[1:]))
```

```
    if isquantifier(s[0]):
        var=gett(s[1:])
        if not(var[0]=="var"): return ""
        return restf(restt(s[1:]))
    if s[0]==":": return restt(s)
    return ""

def testt(s):return displayt(gett(s))

def testf(s):return displayf(getf(s))

r'''
```

2.3 Substitution functions

In this block of code various kinds of substitution are addressed.

There are two kinds of substitution that are made: one is replacement of all occurrences of variables bound by a particular binder [so here we are actually replacing an occurrence, and its index is part of the data supplied], and one is replacement of all free occurrences of variables in a context [occurrence index not relevant].

In all kinds of substitution, the term replacing the variable has its occurrence indices displaced out of the range of indices already used, separately for each time the term is substituted in. This enforces the fact that occurrences of variables are identified exactly if they are bounded by the same binder. Notice that the displacement mechanism preserves identity of occurrences bound in the substituted term by the same binder.

There is also a mechanism for replacing a term uniformly by a variable where it occurs, used by higher order matching.

The function for generating a variable with a new name is included here.

This is a good time to notice a strange feature of this prover. It takes no visible precautions against bound variable capture. Substitution of a free variable into a binding scope with the same shape as the binding variable is possible (though deviously), but the prover will not be confused, because of the invisible occurrence indices. Probably I ought to do something about this.

```
'''

# the range of occurrence indices in a term or formula
# needed for substitution

# I need to think about what
# to do with expressions which actually have
# no variables in them,
# possible if we have defined constants.
```

```
def occt(t):
    if t[0]=="var":  return [t[2],t[2]]
    if t[0]=="set":
        R=occf(t[3])
        return [min(t[2],R[0]),max(t[2],R[1])]
    if t[0]=="let" and t[3][0]=="defined": return occt(t[2])
    if t[0]=="let": return occt(t[2])
    return [newint,newint]

def occf(f):
    if isrelation(f[0]):
        a=occt(f[1])
        b=occt(f[2])
        if f[1][0]=="defined":  return b
        if f[2][0]=="defined":  return a
        return [min(a[0],b[0]),max(a[0],b[0])]
    if f[0]=="~":  return occf(f[1])
    if isconnective(f[0]):
        a=occf(f[1])
        b=occf(f[2])
        return [min(a[0],b[0]),max(a[0],b[0])]
    if isquantifier(f[0]):
        a=occf(f[3])
        return [min(f[2],a[0]),max(f[2],a[1])]
    if f[0]=="let" and f[3][0]=="defined":  return occt(f[2])
    if f[0]=="let":  return occt(f[2])+occt(f[3])
    #use of occt is not an error
    return [newint,newint]

#this is hard substitution for variable occurrences.
# The same copy is put in everywhere (with occurrences refreshed).
#this preserves stratification.  I am not sure whether
# soft substitution based just on surface form of variables is needed at all.
#probably it is, but it should be used sparingly. (setunknown uses it).

#amusingly, this has no compensation
# for bound variable collision because
# (though it can *look* as if it happens) it doesnt.
#but there should be a display function tweak
#to highlight apparent bound variable collisions.  (there isnt yet.)

# these functions compute the range of occurrence indices in a term,
# so that substitution knows how far up to displace occurrence indices
# when substituting a term into a context, and how much to increase nextint
# (the variable containing the next occurrence index)
```

```
# start here, working on definition syntax

def displaceocct(t,d):
    if t[0] == "var":
        return ["var",t[1],t[2]+d]
    if t[0] == "set":
        return ["set",t[1],t[2]+d,
                displaceoccf(t[3],d)]
    if t[0] == "let":
        return ["let", t[1],
                displaceocct(t[2],d),
                displaceocct(t[3],d)]
    if t[0] == "defined":  return t

def displaceoccf(f,d):

    if isrelation(f[0]):
        return [f[0],
                displaceocct(f[1],d),
                displaceocct(f[2],d)]
    if f[0]=="~":  return ["~",
                           displaceoccf(f[1],d)]
    if isconnective(f[0]):
        return [f[0],
                displaceoccf(f[1],d),
                displaceoccf(f[2],d)]
    if isquantifier(f[0]):
        return [f[0],f[1],f[2]+d,
                    displaceoccf(f[3],d)]
    if f[0] == "let":
        return ["let", f[1],
                displaceocct(f[2],d),
                displaceocct(f[3],d)]

# substitution without displacement
# for a variable with a precise occurrence
# index (as with a bound variable)

def subs1t(v,n,t,u):

    if u == ["var",v,n]: return t
    if u[0]=="var":  return u
    if (u[0]=="set"
        and u[1]==v
        and u[2]==n):  return u
    if u[0]=="set":
```

```
        return ["set",u[1],u[2],
                subs1f(v,n,t,u[3])]
    if u[0]=="defined": return u
    if u[0]== "let":
        return ["let",u[1],
                subs1t(v,n,t,u[2]),
                subs1t(v,n,t,u[3])]

def subs1f(v,n,t,u):

    if isrelation(u[0]):
        return [u[0],
                subs1t(v,n,t,u[1]),
                subs1t(v,n,t,u[2])]
    if u[0]=="~":
        return ["~",subs1f(v,n,t,u[1])]
    if isconnective(u[0]):
        return [u[0],
                subs1f(v,n,t,u[1]),
                subs1f(v,n,t,u[2])]
    if (isquantifier(u[0])
        and u[1]==v and u[2]==n):
        return u
    if isquantifier(u[0]):
        return [u[0],u[1],u[2],
                subs1f(v,n,t,u[3])]
    if u[0]== "let":
        return ["let",u[1],
                subs1t(v,n,t,u[2]),
                subs1t(v,n,t,u[3])]

# substitution without displacement
# (soft) for a free variable
# it is used by commands
# for global substitution into proofs, which
# supply the needed displacement
# of the substituted text.

# added fresh displacement
# in each substitution.  This removes
# a technical stratification problem.

def freesubs1t(v,t,u):
    global newint
    if u[0]=="var" and u[1]==v:
        a=occt(t)
```

```
        d=newint-a[0]+1
        newint=a[1]+d
        T=displaceocct(t,d)
        return T
    if u[0]=="var":  return u
    if u[0]=="set" and u[1]==v:  return u
    if u[0]=="set":
        return ["set",u[1],u[2],
                freesubs1f(v,t,u[3])]
    if u[0]=="defined": return u
    if u[0]== "let":
        return ["let",u[1],
                freesubs1t(v,t,u[2]),
                freesubs1t(v,t,u[3])]

def freesubs1f(v,t,u):
    if isrelation(u[0]):
        return [u[0],
                freesubs1t(v,t,u[1]),
                freesubs1t(v,t,u[2])]
    if u[0]=="~":
        return ["~",freesubs1f(v,t,u[1])]
    if isconnective(u[0]):
        return [u[0],
                freesubs1f(v,t,u[1]),
                freesubs1f(v,t,u[2])]
    if isquantifier(u[0]) and u[1]==v: return u
    if isquantifier(u[0]):
        return [u[0],u[1],u[2],
                freesubs1f(v,t,u[3])]
    if u[0]== "let":
        return ["let",u[1],
                freesubs1t(v,t,u[2]),
                freesubs1t(v,t,u[3])]

#these are the real (hard) substitution functions:
#the substituted text has all its occurrence indices made fresh, so
# nothing in it cannot be bound by something outside it,
#and no stratification problems can be introduced
# if the original formula was weakly stratified
# in our very weak sense.

# it feels really weird to write a substitution function
#which completely ignores the variable capture problem.

# probably the display function should alert the user
```

```
#about apparently captured variables.  But this might actually
# be expensive.

# the only user command which would cause apparent variable capture is
# setunknown, and so far I havent installed any warning.  I should do
# a test that it actually behaves as I expect.  (Not true, one can use
# the membership sequent rules to demonstrate apparent variable capture).

# Apparent variable capture is a thing, and to all appearances
#the system behaves correctly;  it doesnt *show*
# you the occurrence data.

def subst(v,n,t,u):
    global newint
    a=occt(t)
    d=newint-a[0]+1
    newint=a[1]+d
    T=displaceocct(t,d)
    return subs1t(v,n,T,u)


def subsf(v,n,t,u):
    global newint
    a=occt(t)
    d=newint-a[0]+1
    newint=a[1]+d
    T=displaceocct(t,d)
    return subs1f(v,n,T,u)

def raisef(f):
    global newint
    a=occf(f)
    d=newint-a[0]+1
    newint=a[1]+d
    F=displaceoccf(f,d)
    return F

# functions for changing variable names to fresh names.
# new names will be generated using priming if necessary.

def dropitem(v,L):
    if L==[]:  return []
    if L[0]==v:  return dropitem(v,L[1:])
    return [L[0]]+dropitem(v,L[1:])

# this function makes the generation of fresh variables much
```

```
# less cluttering.  In effect, binary notation in the primes ' and *

def nextvarname(s):
    if s[0]=="#" or s[0]=="@":
        return nextvarname(s[1:])
    if s[-1]=="'":
        return s[0:-1]+"*"
    if s[-1]=="*":
        return (nextvarname(s[0:-1]))+"'"
    return s+"'"

# this simple function completely replaces renamevar*

def newvariable(v):
    global variables
    if v[1] in variables:
        return newvariable(["var",
                            nextvarname(v[1]),v[2]])
    variables=variables+[v[1]]
    return v

def newvariable2(v):
    global variables
    if v[1] in variables:
        return newvariable2(["var",
                             nextvarname(v[1]),v[2]])
    #variables=variables+[v[1]]
    return v

# this is the complicated term renaming case needed for
# higher order matching.  In the term t, the term v is replaced with the
# new variable w.  It has a peculiar name due to the history of the code.

def renamevart2a(v,w,t):
    global variables
    if (w[1] in variables):
        return renamevart2(v,newvariable2(w),t)
    if (v[0]=="var" and t[0]== "var"
        and t[1]==v[1]):  return [w,w]
    if (not(v[0]=="var") and
        ((not(t[0]=="var"))
         or (t[0]=="var"
            and findunknown(t[1])=="error")
         and equalt(v,t))):
        return [w,w]
```

```
    if t[0]=="var":  return [t,w]
    if (t[0]=="set" and v[0]=="var"
        and t[1]==v[1]):  return [t,w]
    if t[0]=="set":
        return [["set",t[0],t[1],
                 renamevarf2a(v,w,t[3])[0]],w]
    if t[0]=="defined":  return [t,w]
    if t[0]=="let":
        return [["let",t[1],
                 renamevart2a(v,w,t[2]),
                 renamevart2a(v,w,t[3])],w]

def renamevarf2a(v,w,f):
    global variables
    if (w[1] in variables):
        return renamevarf2(v,newvariable2(w),f)
    if isrelation(f[0]):
        return[[f[0],
                renamevart2a(v,w,f[1])[0],
                renamevart2a(v,w,f[2])[0]],w]
    if f[0]=="~":
        return [[f[0],
                 renamevarf2a(v,w,f[1])[0]],w]
    if isconnective(f[0]):
        return[[f[0],
                renamevarf2a(v,w,f[1])[0],
                renamevarf2a(v,w,f[2])[0]],w]
    if (isquantifier(f[0])
        and v[0] == "var" and f[1]==v[1]):
        return [f,w]
    if isquantifier(f[0]):
        return [[f[0],f[1],f[2],
                 renamevarf2a(v,w,f[3])[0]],w]
    if f[0]=="let":
        return [["let",f[1],
                 renamevart2a(v,w,f[2]),
                 renamevart2a(v,w,f[3])],w]

def renamevart2(v,w,t):
    global variables
    global newint
    newint=newint+1
    w=["var",w[1],newint]
    R=renamevart2a(v,w,t)
    variables = variables+[w]
    return R
```

```
def renamevarf2(v,w,t):
    global variables
    global newint
    newint=newint+1
    w=["var",w[1],newint]

    R=renamevarf2a(v,w,t)
    variables = variables+[w]
    return R

r'''
```

2.4 The graph based stratification algorithm

The stratification checker proceeds by first building a weighted directed graph involving all occurrences of variables in the term or formula and base nodes generated for each term involved.

The graph for an atomic sentence has edges in both directions with weights appropriate for the relations: if the formula is $t = u$ the graph consists of the graph for t, the graph for u, and weight 0 edges in both directions between their base nodes. If the formula is $t \in u$ the graph consists of the graph for t, the graph for u, and edges between the base nodes of weight 1 from t to u and -1 from u to t.

Graphs for logically connected and quantified sentences are simply graphs of appropriate subformulas or in the case of binary connectives the union of two graphs.

The graphs for defined formulas are computed by computing the graphs for terms substituted in then linking their base nodes with weights determined by comparing the types of the keys for which they are substituted. Defined terms and formulas are required to be uniquely typed (up to uniform displacement): they have to be not just stratified but connected.

The graph for a variable has a base node with weight 0 links in both directions to the variable occurrence.

The graph for a set $\{x \mid \phi\}$ has graph consisting of the graph for ϕ with a base node with a weight 1 link from the bound occurrence of x and a weight -1 link to the bound occurrence of x.

The graph for a defined term consists of graphs for terms substituted in with links from base nodes of those terms to the base node of the whole term and back with weights determined by the types of the keys in the definition relative to the type of the whole term defined.

The condition for stratification is simple: for any two nodes in the graph, there is at most one length for a path between them (the length of a path being the sum of the weights of the edges in it). The existence of a cycle of nonzero length is exactly equivalent to failure of stratification.

The algorithm used is similar to Dijkstra's algorithm. Choose a starting point in the graph (in the case of a term, this will be the base node). Set the distance from the starting node to itself to 0 and all other distances to ∞.

When we process a node v which has a finite distance from the starting node, we use each edge from v to a node w to compute a distance from the starting node to w. If the distance to w was heretofore infinite, correct it and record a path from the starting node to w. If the distance to w was heretofore finite and the new estimate agrees with it exactly, do nothing. If it disagrees, report failure of stratification and construct a cycle of nonzero length to report as data with the stratification failure conclusion.

In the master algorithm we process each term with finite distance from the starting node once, and we stop when there are no more nodes with finite distance and report a distance function, if no failure of stratification has occurred.

The criterion for stratification here is that each variable occurrence which is connected to the starting variable (which may be taken to be the binding variable in a set abstract) is assigned a consistent finite type. Note that different occurrences of free variables may be assigned different types. The parser tests set abstracts and does not even allow unstratified ones to be displayed. The definition facility requires that the body of a definition be stratified and that no node be at infinite distance from the starting point (stratified and connected), and further that variables of the same shape, even if free, are always assigned the same type independent of occurrence index (so quite rigidly stratified).

```
'''
# constructing the graph from a formula

def addkey(s,x,L):
    if L==[]:return [[s,[x]]]
    if L[0][0]==s:return [[L[0][0],[x]+L[0][1]]]+L[1:]
    return [L[0]]+(addkey(s,x,L[1:]))

# I have done easy tests of stratification, but a full test of
# the type differentials involving set abstracts is probably a good idea.


def graphf(L):
    global countbase
    startgraph=[]
    if isrelation(L[0]) and isterm(L[1]) and isterm(L[2]):
        c1=[str(countbase+1),-1]
        G1=grapht(L[1])
        c2=[str(countbase+1),-1]
        G2=grapht(L[2])
        startgraph=(G1+G2)
        if L[0]=="=":
```

```
        startgraph=addkey(c1,[c1,c2,0],startgraph)
        startgraph=addkey(c2,[c2,c1,0],startgraph)
    if L[0]=="e":
        startgraph=addkey(c1,[c1,c2,1],startgraph)
        startgraph=addkey(c2,[c2,c1,-1],startgraph)
    return startgraph


if L[0]=="~":  return graphf(L[1])
if isconnective(L[0]):  return (graphf(L[1])
                                +graphf(L[2]))
if isquantifier(L[0]):  return graphf(L[3])
if L[0]=="let" and L[3][0]=="defined":
    thedefquery=findvalues(L[3][1],formuladefs)
    if thedefquery==[]:
        print ("undefined definition formula")
        startgraph= addkey([basename,-1],
                           [[basename,-1],
                            [basename,-1],1],
                           startgraph)
        return startgraph
    thetypes=thedefquery[0][1]
    thetypequery=findvalues(L[1][1],thetypes)
    if thetypequery==[]:
        print ("no type for this variable found")
        startgraph= addkey([basename,-1],
                           [[basename,-1],
                            [basename,-1],1],
                           startgraph)
        return startgraph
    thetype=thetypequery[0]
    countbase=countbase+1
    c2=[str(countbase),-1]
    c1=[str(countbase+1),-1]
    G1=grapht(L[2])
    startgraph=G1
    startgraph=addkey(c2,[c2,c1,thetype],
                      startgraph)
    startgraph=addkey(c1,[c1,c2,-thetype],
                      startgraph)
    return startgraph
if L[0]=="let":
    search=L[3]
    while(not(search[0]=="defined")):
        search=search[3]
    thedefquery=findvalues(search[1],formuladefs)
```

```
        if thedefquery==[]:
            print ("undefined definition term")
            startgraph= addkey([basename,-1],
                               [[basename,-1],
                                [basename,-1],1],
                               startgraph)
            return startgraph
        thetypes=thedefquery[0][1]
        thetypequery=findvalues(L[1][1],thetypes)
        if thetypequery==[]:
            print ("no type for this variable found")
            startgraph= addkey([basename,-1],
                               [[basename,-1],
                                [basename,-1],1],
                               startgraph)
            return startgraph
        thetype=thetypequery[0]
        c2=[str(countbase+1),-1]
        G2=graphf(L[3])
        c1=[str(countbase+1),-1]
        G1=grapht(L[2])

        startgraph=G1+G2
        startgraph=addkey(c2,[c2,c1,thetype],
                          startgraph)
        startgraph=addkey(c1,[c1,c2,-thetype],
                          startgraph)
        return startgraph



    return []

#  adding in a node [base,-1] standing for the term itself in grapht

# this means there are lots of base nodes generated, one for every subterm!

def grapht0(L,basename):
    startgraph=[]
    if L[0]=="var":
        startgraph=addkey([basename,-1],
                          [[basename,-1],
                           [L[1],L[2]],0],
                          startgraph)
        startgraph=addkey([L[1],L[2]],
                          [[L[1],L[2]],
```

```
                      [basename,-1],0],
                     startgraph)
    return startgraph
if L[0]=="set":
    startgraph=graphf(L[3])
    startgraph=addkey([basename,-1],
                     [[basename,-1],
                      [L[1],L[2]],-1],
                     startgraph)
    startgraph=addkey([L[1],L[2]],
                     [[L[1],L[2]],
                      [basename,-1],1],
                     startgraph)
    return startgraph
if L[0]=="defined":
    startgraph=[]
    startgraph=addkey([basename,-1],
                     [[basename,-1],
                      [basename,-1],0],
                     startgraph)
    return startgraph
if L[0]=="let":
    c1=[basename,-1]
    G1=grapht0(L[3],basename)
    c2=[str(countbase+1),-1]
    G2=grapht(L[2])

    startgraph = G1+G2
    search=L[3]
    while(not(search[0]=="defined")):
        search=search[3]
    thedefquery=findvalues(search[1],termdefs)
    if thedefquery==[]:
        print ("undefined definition term"
               +(search[1]))
        startgraph= addkey([basename,-1],
                           [[basename,-1],
                            [basename,-1],1],
                           startgraph)
        return startgraph
    thetypes=thedefquery[0][1]
    thetypequery=findvalues(L[1][1],thetypes)
    if thetypequery==[]:
        print ("no type for this variable found")
        startgraph= addkey([basename,-1],
                           [[basename,-1],
```

```
                                [basename,-1],1],
                               startgraph)
            return startgraph
        thetype=thetypequery[0]
        startgraph=addkey(c1,[c1,c2,thetype],
                          startgraph)
        startgraph=addkey(c2,[c2,c1,-thetype],
                          startgraph)
    return startgraph

countbase=1

def grapht(L):
    global countbase
    countbase=countbase+1
    return grapht0(L,str(countbase))




# using graph method to check stratification

# with the graph as input
# and a starting point, compute a distance function
# (a relative type assignment) from that starting point.

# a distance function is a list of pairs of a vertex
# and either the empty set (for infinite distance)
# or a singleton of an integer, followed by
# either the empty set or a path (for data
# extraction)

# make the initial distance table
# from a given starting point

def makedistances(L,x):
    if L==[]:  return []
    m=makedistances(L[1:],x)
    if not(getdistance(m,L[0][0])=="error"):
        return m
    if L[0][0]==x:  return [[x,[0],[x]]]+m

    return ([[L[0][0],[],[]]]
            +makedistances(L[1:],x))
```

```
def makedistancet(L):
    return makedistances(L,L[0][0])

# get the distance from the starting point
# to v (as currently estimated)

def getdistance(L,v):

    if L==[]:  return "error"
    if L[0][0]==v:
        if L[0][1]==[]:return "infty"
        return L[0][1][0]
    return getdistance(L[1:],v)

#get the path from the starting point
# to V (if we have one)

def getpath(L,v):

    if L==[]:
        print ("error")
        return []
    if L[0][0]==v:
        if L[0][2]==[]:
            print ("none")
            return []
        return L[0][2]
    return getpath(L[1:],v)

# drop the distance to v (in order to replace it)

def dropkey(L,v):
    if L==[]:  return []
    if L[0][0]==v:  return L[1:]
    return [L[0]]+dropkey(L[1:],v)

# function for updating a distance function
# given an edge to work with

# reverse a list, needed in
# updatedistances to display the bad cycle nicely

def rev(L):
    if L==[]:  return []
    return rev(L[1:])+[L[0]]
```

```
def rev2(L):
    if L==[]:  return []
    return rev2(L[1:]) +[rev(L[0])]

def updatedistances(L,v,w,n):


    vdist = getdistance(L,v)

    if not(type(vdist)==int):  return "error condition"
    wdist = getdistance(L,w)

    # if w already has a different distance estimate,
    #construct and return a cyle of nonzero length

    if not(wdist=="infty" or wdist==vdist+n):
        return["stratification failure",
               (getpath(L,v)+[[v,n,w]]
                +rev2(getpath(L,w)))]

    # if the distance estimate is new,
    #add it (nothing to do if no updating needed)

    if wdist=="infty":
        return ([[w,[vdist+n],
            getpath(L,v)+[[v,n,w]]]]
                +dropkey(L,w))
    # this output is now a bit more readable
    # mod the number of new nodes.

    return L



def stratify(G,x):

    #initialize the distances

    thedistances=makedistances(G,x)

    i=0
    while(i<len(G)):

       # move thus far uninformative vertices to the end
       j=i
       while(j<len(G)
```

```
              and getdistance(thedistances,G[i][0])
              =="infty"):
             G=G[0:i]+G[i+1:]+[G[i]]
             j=j+1

        #done if nothing informative is found


        if getdistance(thedistances,G[i][0])=="infty":
            return thedistances
        k=0
        while(k<len(G[i][1])):
            thedistances = updatedistances(thedistances,
                                           G[i][1][k][0],
                                           G[i][1][k][1],
                                           G[i][1][k][2])
            #in the event of stratification failure
            #return the bad cycle
            if thedistances[0]=="stratification failure":
                return thedistances
            k=k+1
        i=i+1
    # if no errors occur, return the distance table
    # (scheme of relative types)
    return thedistances

# for the moment I only have a stratification tester for formulas.  But
# set abstracts are tested by the parser.  So testt will test any nontrivial term.
# this is outdated:  there is now a function for terms as well.

def strattest(s):
    t=getf(s)
    g=graphf(t)
    v=g[0][0]
    return stratify(g,v)

# tester for terms

def strattest2(s):
    t = gett(s)
    b=countbase+1
    g = grapht(t)
    return(stratify(g,[str(b),-1]))

r'''
```

2.5 The definition mechanism

The basic idea of defined terms and formulas is to introduce an abbreviation (a possibly multiply primed capital letter) for a stratified connected term or formula (in which all variables of the same shape have the same type and every variable is assigned a uniquely determined type) and to access it by assigning term values to variables appearing in it. We suggest *definiendum* for the possibly primed capital letter and *definiens* for the stratified connected term or formula which replaces it, as handy terminology for components of this construction.

Although the notation does not express this, the internal variables are of the form `#v` and so cannot be entered by the user. They will also not be displayed: any free variables in the definition are converted to `@v` form. Thus no variable capture problems are incurred by the repeated application of substitutions involved in evaluating a definiendum with several arguments.

Evaluation of a defined term with substitution suffixes (prefixes in the input notation) is carried out as a cascaded sequence of substitutions after renaming variables with # as needed.

The `deft` and `deff` user commands introduce defined terms and formulas respectively. When a definition is displayed, stratification information about the definiens is shown. The `showdeft` and `showdeff` functions allow them to be viewed.

NOTE: the official criterion for a definiens is that it be stratified and connected. But in fact it must contain at least one variable: a definiens consisting entirely of defined notions is reported as unstratified and disconnected for the moment. Replacing a definiens Δ with $\{x \mid x \in \Delta\}$ should fix this cheaply.

```
,,,


# definition mechanism

# return a simplified table of relative types
# which now omits the clutter of base nodes
# and also requires a unique type for each variable shape

def findvalues(v,L):
    if L==[]:  return []
    if len(L[0])<2:
        print ("Key list error condition")
        return []
    rest = findvalues(v,L[1:])
    if L[0][0]==v:  return [L[0][1]]+dropitem(L[0][1],rest)
    return rest

def collapsestrat1(L):
```

```
    M=L
    L2=[]
    while (not (M==[])):
        var=M[0][0][0]
        if M[0][1]==[]: return []
        #we want connected
        val=M[0][1][0]
        vals = findvalues(var,L2)
        if vals ==[]: L2=L2+[[var,val]]
        if not(vals == [] or vals ==[val]):
            return [] #conflict of values
        M=M[1:]
    return L2

def collapsestrat2(L):
    if L==[]:  return L
    if "0" <= L[0][0][0] and L[0][0][0]<= "9":
        return collapsestrat2(L[1:])
    return [L[0]]+collapsestrat2(L[1:])

def collapsestrat(L):
    return collapsestrat2(collapsestrat1(L))

def ctest(s):
    return collapsestrat(strattest(s))

termdefs=[]

def showtermdefs():

    print(termdefs)

# USER COMMAND

# introduce a term definition

def deft(key,term):
    global termdefs



    K= gett(key)
    if not (K[0]=="defined"):
        print ("Cant define that")
        return "Cant define that"
    if not(findvalues(displayt(K),termdefs)==[]):
```

```
        print("Symbol already defined")
        return "Symbol already defined"
    T=gett(term)
    if not(isterm(T)):
        print ("Term entry error")
        return "Term entry error"
    c=[str(countbase+1),-1]
    TT=grapht(T)

    L=collapsestrat(stratify(TT,c))
    if L==[]:
        print("Term is unstratified or disconnected")
        return "Term is unstratified or disconnected"
    usercommand("deft "+'("'+key+'", "'+term+'")\n')

    termdefs=[[displayt(K),[T,L]]]+termdefs
    print ((displayt(K))+ " = "+(displayt(T)))
    print (L)

# USER COMMAND

#display a term definition

def showdeft(s):
     if findvalues(s,termdefs)==[]:
         print ("Undefined term symbol")
         return "Undefined term symbol"
     print (s+" = "
            +(displayt
              (findvalues(s,termdefs)[0][0])))
     print (findvalues(s,termdefs)[0][1])
     return "Done!"

# USER COMMAND

#display a formula definition

def showdeff(s):
     if findvalues(s,formuladefs)==[]:
         print ("Undefined formula symbol")
         return "Undefined formula symbol"
     print (s+" = "
            +(displayf
              (findvalues(s, formuladefs)[0][0])))
     print (findvalues(s,formuladefs)[0][1])
     return "Done!"
```

```
formuladefs=[]

def showformuladefs():

    print(formuladefs)

# USER COMMAND

# introduce a formula definition

def deff(key,term):
    global formuladefs

    K= gett(key)
    if not (K[0]=="defined"):
        print ("Cant define that")
        return "Cant define that"
    if not(findvalues(displayt(K),termdefs)==[]):
        print("Symbol already defined")
        return "Symbol already defined"
    T=getf(term)
    if not(isformula(T)):
        print ("Formula entry error")
        return "Formula entry error"
    c=[str(countbase+1),-1]
    TT=graphf(T)

    L=collapsestrat(stratify(TT,TT[0][0]))
    if L==[]:
        print("Term is unstratified or disconnected")
        return "Term is unstratified or disconnected"
    usercommand('deff '+'("'+key+'", "'+term+'")\n')
    formuladefs=[[displayt(K),[T,L]]]+formuladefs
    print ((displayt(K))+ " = "+(displayf(T)))
    print (L)

# definition expansion involves
# quite elaborate variable reshuffling

def defexpandt(t):
    # if applied to a definition term, simply expand it
    global newint
    if t[0]=="defined":
        return deatifyt(atify(t))
    if t[0]=="let":
```

```
        T=atify(t)
        U=cascadesubst(T)
        return (deatifyt(U))
    return t

def defexpandf(t):

    if t[0]=="let":
        T=atify2(t)
        U=cascadesubsf(T)
        return (deatifyf(U))
    return t

# Note that for variable security, the internal variables
# created by atify are now hashed (#).  These variables must
# not be possible for the user to create in displayed text in any way.

def atify(t):
    global variables
    global newint
    if t[0]=="let":
        if not "#"+t[1][1] in variables:
            variables = variables+["#"+t[1][1]]
        return [t[0],["var",
                      "#"+t[1][1],t[1][2]],t[2],
                 atify(t[3])]

    if t[0]=="defined":
             if findvalues(t[1],termdefs) == []:
                print ("undefined symbol "+t[1]
                       +" in atify")
                return ("undefined symbol in atify")
             u=findvalues(t[1],termdefs)[0][0]
             a=occt(u)
             d=newint-a[0]+1
             newint=a[1]+d
             T=displaceocct(u,d)
             return atify2t(T)

def atify2(t):
    global variables
    global newint
    if t[0]=="let":
        if not "#"+t[1][1] in variables:
            variables = variables+["#"+t[1][1]]
        return [t[0],["var",
```

```
                    "#"+t[1][1],t[1][2]],
               t[2],atify2(t[3])]

    if t[0]=="defined":
            if findvalues(t[1],formuladefs) == []:
               print ("undefined symbol "
                      +t[1]+"  in atify2")
               return ("undefined symbol "
                       +t[1]+"  in atify2")
            u=findvalues(t[1],formuladefs)[0][0]
            a=occf(u)
            d=newint-a[0]+1
            newint=a[1]+d
            T=displaceoccf(u,d)

            return atify2f(T)

def atify2t(t):
    global variables
    if t[0]=="var":
        if not "#"+t[1] in variables:
            variables = variables+["#"+t[1]]
        return ["var","#"+t[1],t[2]]
    if t[0]=="set":
        if not "#"+t[1] in variables:
            variables = variables+["#"+t[1]]
        return [t[0],"#"+t[1],t[2],
                atify2f(t[3])]
    if t[0]=="defined":  return t
    if t[0]=="let":
        return [t[0],t[1],
                atify2t(t[2]),
                atify2t(t[3])]

def atify2f(f):
    global variables
    if isrelation(f[0]):
        return [f[0],
                atify2t(f[1]),
                atify2t(f[2])]
    if isconnective(f[0]):
        return [f[0],
                atify2f(f[1]),
                atify2f(f[2])]
    if f[0]=="~":  return [f[0],
                           atify2f(f[1])]
```

```
    if isquantifier(f[0]):
        if not "#"+f[1] in variables:
            variables = (variables
                         +["#"+f[1]])
        return [f[0],"#"+f[1],f[2],
                atify2f(f[3])]
    if f[0]=="let":
        return [f[0],f[1],
                atify2t(f[2]),
                atify2t(f[3])]

# It is an accident of the thought process in correcting for
# variable security that hashify actually adds @

def hashifyt(t):
    global variables
    if t[0]=="var":
        if not "@"+t[1] in variables:
            variables = variables+["@"+t[1]]
        return ["var","@"+t[1],t[2]]
    if t[0]=="set":
        if not "@"+t[1] in variables:
            variables = variables+["@"+t[1]]
        return [t[0],"@"+t[1],t[2],
                hashifyf(t[3])]
    if t[0]=="defined":  return t
    if t[0]=="let":  return [t[0],t[1],
                             hashifyt(t[2]),
                             hashifyt(t[3])]

def hashifyf(f):
    global variables
    if isrelation(f[0]):
        return [f[0],
            hashifyt(f[1]),
            hashifyt(f[2])]
    if isconnective(f[0]):
        return [f[0],
            hashifyf(f[1]),
            hashifyf(f[2])]
    if f[0]=="~":
        return [f[0],hashifyf(f[1])]
    if isquantifier(f[0]):
        if not "@"+f[1] in variables:
            variables = variables+["@"+f[1]]
        return [f[0],"@"+f[1],f[2],hashifyf(f[3])]
```

```
    if f[0]=="let":  return [f[0],f[1],
                             hashifyt(f[2]),
                             hashifyt(f[3])]


def cascadesubst(t):
    if t[0] == "let" and t[1][1][0] =="#":
        return freesubs1t(t[1][1],t[2],
                          cascadesubst(t[3]))
    return t

def cascadesubsf(t):
    if t[0] == "let" and t[1][1][0] =="#":
        return freesubs1f(t[1][1],t[2],
                          cascadesubsf(t[3]))
    return t

# actually replaces # with @

def removeat(s):
    global variables
    if s[0]=="#":
        if not "@"+s[1:] in variables:
            variables=variables+["@"+s[1:]]
        return "@"+s[1:]
    return s

# change all hashes to ats (so the name is funny).

def deatifyt(t):
    if t[0]=="var":
        return ["var",
                removeat(t[1]),t[2]]
    if t[0]=="set":
        return [t[0],removeat(t[1]),t[2],
                deatifyf(t[3])]
    if t[0]=="defined":  return t
    if t[0]=="let":
        return [t[0],t[1],
                deatifyt(t[2]),
                deatifyt(t[3])]

def deatifyf(f):
    if isrelation(f[0]):
        return [f[0],
                deatifyt(f[1]),
```

```
                deatifyt(f[2])]
    if isconnective(f[0]):
        return [f[0],
                deatifyf(f[1]),
                deatifyf(f[2])]
    if f[0]=="~":
        return [f[0],deatifyf(f[1])]
    if isquantifier(f[0]):
        return [f[0],removeat(f[1]),f[2],
                deatifyf(f[3])]
    if f[0]=="let":
        return [f[0],f[1],deatifyt(f[2]),
                deatifyt(f[3])]

r'''
```

2.6 The sequent prover (with complete user command list for the software as a whole)

The remainder of the work is development of a user driven sequent prover for NF (which could be adapted to a prover for SF [for which the cut elimination result exists] or NFU by changing the right equality rule). This parallels my earlier work on versions of the Marcel prover, but it does have interesting features.

The general plan of the sequent prover is that one uses `start s` to introduce a sequent of the form

$$\frac{}{P}$$

where P is the result of parsing the string s.
One then has a limited number of commands as options.

- The `right()` command applies an appropriate right rule to the first proposition on the right (below the line that is). It will sometimes expand a defined notion if there is nothing else to do.
- The `left()` command applies an appropriate left rule to the first proposition on the left (above the line, that is). It will sometimes expand a defined notion if there is nothing else to do.
- The `getleft(n)` command brings the nth proposition on the left (above the horizontal bar, actually) to the front.
- The `getright(n)` command brings the nth proposition on the right to the front.

- The `pruneleft(n)` and `pruneright(n)` commands are as `getleft(n)` and `getright(n)` except that they bring the designated proposition to the front of the appropriate side of the sequent, then remove it. This can be used with care to declutter an argument.

- The `done()` command invites the prover to recognize that the current sequent is an axiom (that the propositions at the beginnings of the lists on left and right are the same). Identity of propositions up to α-equivalence, ignoring occurrence data, is supported.

  The `done()` command is dynamic: its underlying equality test will make an equation between an unknown variable in one term or formula and the matching term in the other term or formula true by executing the `setunknown` command on the fly. It does this safely (it tests that the `setunknown` command will execute without error). A static equality test will probably be needed eventually, but `done` is currently its only client.

  A further, even more dangerous feature has been added. When the equality function checks whether $x \in u?$ matches $P(x)$ (where $u?$ is an unknown and $P(x)$ is any stratified formula in x, it will attempt to set the unknown $u?$ to the value $\{x \mid P(x)\}$. It does (mod bugs) check that this can actually be done. This is a kind of higher order matching. It can be very nice for forestalling the need to manually enter values to be assigned to unknowns.

  I have also added reflexivity of equality as an axiom, and the `done()` command handles this. This may have dynamic effects if the prover can make the terms in an equation equal by setting unknowns suitably. The other rules prove reflexivity of equality, but this should remove lots of junk lines from proofs.

  Where the `done` command fails, one might want to use `back()` to restore the previous state, because the prover might have made undesired global substitutions for unknowns.

- The `setunknown(u,t)` command allows one to set the value (globally in the proof) of an "unknown" `u` (a free variable introduced by the right rule for the existential quantifier or the left rule for the universal quantifier) to a term `t`. In a usual presentation, a specific term is supplied when the rule is applied: here the unity of the `left()` and `right()` rules is preserved by separating out the choice of witness. It may also be useful to delay choosing the witness until it is evident from developments in the proof what might work. There are limitations on what can replace an "unknown" [the term replacing it cannot contain any fresh variable introduced by quantifier rules which is introduced after the unknown which is being replaced].

  The command also now allows a value not including any fresh variables to be set for a variable free in the sequent. This gives theorems and definitions with free variables in them some limited usefulness. It provides support for the design decision of using output notation for `u`, since a variable

beginning with an at sign (as a free variable introduced by application of a definition or theorem will) has no input notation.

The user should note that the notation for `t` is input notation: variables in `t` will have to be typed differently!

- The `Cut(f)` command executes the cut rule with the formula `f`.
- The `varelim()` command uses an equation at the head of the left list of the sequent to make a global substitution in the current sequent for a fresh variable when priority conditions on fresh variables permit this. The equation is then eliminated. The use of this is to reduce variable clutter, which we recall is a problem in large proofs which we encountered in older versions of this system.
- The `deft(key,term)` command introduces a defined term (defining `key`, a possibly primed capital letter, as `term`) and the `deff(key,formula)` command similarly introduces a defined formula. These can further be accessed by assigning values to variables: in the definiens: a defined term can be used as a constant, but a defined formula must have at least one assignment. I need to provide examples.
- The `savetheorem` command saves the current theorem: it takes the name to be given to the theorem as its sole argument.
- The `theoremcut` command has as its argument the name of a theorem, whose conclusion is introduced as a hypothesis in the current sequent. This allows saved theorems to be used (examination of the proofs will reveal that this is a hidden use of cut; one of the sequents in the cut is never seen by the user).
- The `look()` command lets you look at the current sequent.
- The `skip()` command lets you move the current sequent to the end of the work queue and go to the next one. This lets you cycle through all the work to be done and view it (This command is much more sensible than its analogues in older versions, where we used an actual tree rather than a list of sequents as our data structure).
- The `back()` command undoes a command, roughly speaking. A stack of proof states is updated from time to time. Security problems experienced with it earlier seem to be resolved (fingers crossed).
- The `savetheproof(s)` command will save a snapshot of the current proof state with a key given as its argument; `loadproof(s)` command will return to the proof saved with the key given as argument. This allows one to leave a proof and prove needed lemmas to be introduced by `theoremcut`, for example, then return with the lemma ready to use.

- The `setlog` command sets the name of a log file to which user commands will be appended as they are executed. This creates scripts which can be used to reconstruct work done under the prover. I close a log file by setting the log file to a default name (`setlog (``done'')` ). There must be a directory called `logs` in the current directory for a log file to be set up successfully.

- The `demo()` and `undemo()` commands turn demo mode on and off. Running a script in demo mode causes a pause after each command (hit any key to continue).

- The `showdeft`, `showdeff` and `showtheorem` commands display a term or formula definition or theorem named by their string arguments. The `showtheorems()` command pages through all the theorems (hit any key to continue). `showdefts()` and `showdeffs()` are analogous. `showtheproof()` shows the current complete proof all at once, possibly a large object. `LogTheProof()` does the same thing and also posts it to the log.

- `makeconstructive()` and `makeclassical()` toggle between classical and intuitionistic logic. Toggling to intuitionistic logic clears the current proof. Intuitionistic logic is implemented by causing the `left()` and `right()` commands to discard all right propositions in a sequent except the first one *before* applying a rule. Afterward, it might display more than one proposition on the right: use `getright()` to decide which one to keep. Note that the stratified comprehension of this version of Marcel is more liberal than the stratified comprehension used in presentations of intuitionistic NF up to this point.

The prover maintains a stack of sequents needing attention. When the `left()` or `right()` rule is applied, one or two sequents are appended to the end of the proof [which is a list of sequents with pointers (really just lists of line numbers) from sequents to ones justifying them] and added to the top of the stack, and one then proceeds to the first sequent on the stack. Other rules affecting proof state have appropriate effects on the stack.

When every line has a justification, the proof is complete.

The meat of the logic supported is in the functions `leftaction` and `rightaction`. which are used to indicate how to construct the one or two sequents from which a current sequent is derived.

We will present the rules shortly. I have copied these with some editing from a manual for a long-ago version of Marcel. There is a special additional left rule for membership in a variable which is described at the end.

This version of Marcel frankly has NF as its logic. This could be changed straightforwardly. The logic of this version is classical, but it could very easily be made intuitionistic.

In addition to applying the rules stated below, the prover also expands defined terms and formulas in a manner which should be documented. The present

rule is that the prover will expand a defined formula if it is the left or right leading sequent (depending on which command is being applied), and it will expand defined terms in a membership statement appearing as left or right leading sequent, preferring to expand the right term.

```
 '''

#start sequent development

# a proof is a list of sequents

# a sequent contains three parts, a list of left proposition,
# a list of right propositions,
# and a list of positions (integers) of sequents
# in the proof intended to justify it, or [-1] if it is not yet
# proved.  A sequent rule will take
# the left and right propositions and return
# one or two new sequents, inserting them
# at the end of the proof as
# further goals, and adjusting the third component
# of the original sequent
# to point to them.

def displayproplist(L):
    if L==[]: return "\n"
    n=0
    thelist="\n"
    while (n<len(L)):
        thelist = thelist+"\n\n"+str(n)+".  "+displayf(L[n])
        n=n+1
    return thelist

def displaysequent(n,S):

    return ("\n\nLine number "+(str(n))
            +"\n\n"+displayproplist(S[0])+"\n----\n"
            +displayproplist(S[1])+"\n\n"+"by lines "+str(S[2]))


#substitution into a sequent

def subs1s(v,t,S):
    L=S[0]
    L2=[]
    while(not(L==[])):
```

```
        L2=L2+[freesubs1f(v,t,L[0])]
        L=L[1:]
    R=S[1]
    R2=[]
    while(not(R==[])):
        R2=R2+[freesubs1f(v,t,R[0])]
        R=R[1:]
    return [L2,R2,S[2]]


theproof=[]

def displayproof(P):
    if P==[]: return []
    Q=P
    o=""
    n=0
    while(not(Q==[])):
        o=o+displaysequent(n,Q[0])
        Q=Q[1:]
        n=n+1
    print(o)
    return o

# USER COMMAND

def showtheproof():

    print(displayproof(theproof))

#the line number in the proof where we are working

theline=0

def showtheline():

    print(theline)

# something like the original proof graph traversal is now implemented,
#using a stack to indicate which line we want to do next

#this is the stack of lines to be dealt with
#when it is exhausted, simply proceed to the next line
# (if this actually happens)

linestack=[]
```

```
def showlinestack():

    print(linestack)

# the stack of previous proof states maintained for the back() function

proofstack=[]

#

demostate=False

# USER COMMANDS

def demo():
    global demostate
    demostate=True

def undemo():
    global demostate
    demostate=False

# the function which determines whether we are done with the current line,
# and if we are done proceeds to the next line.

def displaynextline():
    global theline
    global linestack
    global proofstack
    global variables
    global unknowns
    global freshvars
    global newint
    global theproof
    global theline
    global unknownassignments
    unknownassignments=[]

    if not (linestack==[]) and not(theproof[theline][2]==[-1]):
        theline = linestack[0]
        linestack=linestack[1:]
        print(displaysequent(theline,theproof[theline]))
        logfile.write('\n"""')
        logfile.write(displaysequent(theline,theproof[theline]))
        logfile.write('\nNext!"""\n\n')
```

```
        print ("Next!")
        proofstack=[[theproof,theline,
                     newint,countbase,
                     variables,freshvars,
                     unknowns,linestack]]+proofstack
        if demostate==True:
            print("Hit any key to continue\n")
            input()

        return("Next!")
    while (theline<len(theproof) and not(theproof[theline][2]==[-1])):
        theline=theline+1
    if theline==len(theproof):
        print ("Done!")
        logfile.write('\n"""Done!"""\n\n')
        proofstack=[[theproof,theline,
                     newint,countbase,
                     variables,freshvars,
                     unknowns,linestack]]+proofstack
        return "Done!"
    print(displaysequent(theline,theproof[theline]))
    logfile.write('\n"""')
    logfile.write(displaysequent(theline,theproof[theline]))
    logfile.write('\nNext!"""\n\n')
    proofstack=[[theproof,theline,
                 newint,countbase,
                 variables,freshvars,
                 unknowns,linestack]]+proofstack
    if demostate==True:
            print("Hit any key to continue\n")
            input()
    print ("Next!")

# USER COMMAND

# save a snapshot of the current proof state

savedproofs=[]

def savetheproof(s):
    global savedproofs
    global theline
    global linestack
    global proofstack
    global variables
    global unknowns
```

```
    global freshvars
    global newint
    global theproof
    global theline
    if not(findvalues(s,savedproofs) == []):
        print ("A proof has already been saved as "+s)
        return "name already used"
    usercommand('savetheproof("'+s+'")\n')
    savedproofs = [[s,[theproof,theline,
                    newint,countbase,
                    variables,freshvars,
                    unknowns,linestack,proofstack]]]+savedproofs

# USER COMMAND

# load a saved proof state

def loadproof(s):

    global savedproofs
    global theline
    global linestack
    global proofstack
    global variables
    global unknowns
    global freshvars
    global newint
    global theproof
    global theline
    if findvalues(s,savedproofs)==[]:
        print("No saved proof state found that is called "+s)
        return "No state found with that name"
    usercommand('loadproof("'+s+'")\n')
    P=findvalues(s,savedproofs)[0]
    theproof=P[0]
    theline=P[1]
    newint=P[2]
    countbase=P[3]
    variables=P[4]
    freshvars=P[5]
    unknowns=P[6]
    linestack=P[7]
    proofstack=P[8]
    look()

# USER COMMAND
```

```
#put the current line at the bottom
# of the stack of sequents to be dealt with
#and bring on the next sequent

def skip():
    global proofstack
    global theline
    global linestack
    if len(linestack)<1:
        print("Nothing to skip")
        displaynextline()
        return "Nothing to skip"
    usercommand("skip()\n")
    L=theline
    theline=linestack[0]
    linestack=linestack[1:]+[L]
    proofstack=[[theproof,theline,
                newint,countbase,
                variables,freshvars,
                unknowns,linestack]]+proofstack
    displaynextline()

# USER COMMAND

# restore the last prover state for which displaynextline() was called.
# Historically a bit buggy.
# it seems to be reliable now.

# it is worth noting that definitions and saved theorems
# are not part of the proof state managed by back().
# it is not clear to me whether this could cause trouble.

def back():

    global theproof
    global theline
    global newint
    global variables
    global freshvars
    global unknowns
    global linestack
    global proofstack
    if proofstack==[]:
        print ("No back information")
        return ("No back information")
    usercommand("back()\n")
```

```
    P=proofstack[0]
    proofstack=proofstack[1:]
    theproof=P[0]
    theline=P[1]
    newint=P[2]
    countbase=P[3]
    variables=P[4]
    freshvars=P[5]
    unknowns=P[6]
    linestack=P[7]
    theproof[theline][2]=[-1]
    look()
    return ("Backed up")

# USER COMMAND

def look():
    if theproof == []:
        print("no proof to show")
        return ("No proof to show")
    if not(theline==len(theproof)):
        print(displaysequent(theline,theproof[theline]))
    if theline==len(theproof):
        print(displaysequent(theline,theproof[0]))
    if theline==len(theproof): print("Done!")



# USER COMMAND
# initialize a sequent to be proved

def start(f):
    global newint
    global theproof
    global theline
    global variables
    global unknowns
    global freshvars
    global proofstack
    global linestack
    global countbase

    newint = 1
    countbase = 1
    variables=[]
    unknowns=[]
```

```
    freshvars=[]
    linestack=[]
    proofstack=[]
    if not(isformula(getf(f))):
        print("Formula entry error")
        return("Formula entry error")
    usercommand('start ("'+f+'")\n')
    theproof=[[[],[getf(f)],[-1]]]
    theline=0
    displaynextline()

# utility to generate the line numbers
# on which proof of a line depends immediately

def nextrefs(n):
    i=0
    r=[]
    l=len(theproof)
    while (i<n):
        r=r+[l+i]
        i=i+1
    return r

# toggle indicating whether constructive logic is to be used

constructive=False

# USER COMMANDS

# toggle constructivity

def makeconstructive():
    global constructive
    global theorems
    if constructive==True:
        print ("Already constructive")
        return "No action taken"
    theorems = []
    start("=xx")
    constructive = True

def makeclassical():
    global constructive
    if constructive==False:
        print("Already classical")
        return "No action taken"
```

```
    constructive = False



# USER COMMAND
# apply a left rule

def left():

    global theproof
    global linestack
    global variables

    if len(theproof[theline][0])==0:
        displaynextline()
        return "No left proposition to act on!"
    L=leftaction(theproof[theline][0][0])
    if L==[[]]:
        displaynextline()
        return "Left action unsuccessful"
    usercommand("left()\n")
    theproof[theline][2]=nextrefs(len(L))
    i=0
    while (i<len(L)):
        A=L[i][0]+theproof[theline][0][1:]
        if (constructive==True
            and not(theproof[theline][1]==[])):
            theproof[theline][1]=[theproof[theline][1][0]]
        B=L[i][1]+theproof[theline][1]
        theproof=theproof+[[A,B,[-1]]]
        i=i+1
    linestack=theproof[theline][2]+linestack
    displaynextline()

# USER COMMAND
# apply a right rule

def right():

    global theproof
    global linestack
    global variables

    if len(theproof[theline][1])==0:
        displaynextline()
        return "No left proposition to act on!"
```

```
    R=rightaction(theproof[theline][1][0])
    if R==[[]]:
        displaynextline()
        return "Right action unsuccessful"
    usercommand("right()\n")
    theproof[theline][2]=nextrefs(len(R))
    i=0
    while (i<len(R)):
        A=R[i][0]+theproof[theline][0]
        if (constructive==True
            and not(theproof[theline][1]==[])):
            theproof[theline][1]=[theproof[theline][1][0]]
        B=R[i][1]+theproof[theline][1][1:]
        theproof=theproof+[[A,B,[-1]]]
        i=i+1
    linestack=theproof[theline][2]+linestack
    displaynextline()

#because of our occurrence based mechanics,
# determining equality of terms takes work.
#everything has a price.

# When terms are not equal and evaluating
# an unknown will make them true, the appropriate
# setunknown action is executed to make them equal.

# we might want a nondynamic equality function,
# but for the moment done() is the only client

r'''
```

2.7 A note on equality

We note that the equality functions for terms and formulas have `done()` as their only client, and they may be found disconcertingly dynamic. These commands will expand definitions if they might allow equality to be established. More alarmingly, they will set values for unknown variables if this might allow equality to be established, both in the direct way of setting `x?` to `t` if they match and dependency relations for unknowns allow it, and more alarmingly yet, setting `x?` to $\{u \mid \phi\}$ if `t e x?` matches ϕ' and the result of replacing `t` with `u` in ϕ' is ϕ.

```
'''
```

```
unknownassignments=[]

def equalt(t1,t2):
    global unknowns
    global unknownassignments
    if t1[0]=="var" and t1[1]==t2[1]:  return True
    if t1[0]=="var" and not(findunknown(t1[1])=="error"):
        V=findunknown(t1[1])
        T=t2
        L = freshvarlistt(T)
        while(not(L==[])):
          if not(L[0] in V[1]):
            return False
          L=L[1:]
        if findvalues(t1[1],unknownassignments)==[]:
            unknownassignments=[[t1[1],t2]]+unknownassignments
            setunknown2(t1[1],t2)
            return True
        return equalt(findvalues(t1[1],unknownassignments)[0],t2)
    if t2[0]=="var" and not(findunknown(t2[1])=="error"):
        V=findunknown(t2[1])
        T=t1
        L = freshvarlistt(T)
        while(not(L==[])):
          if not(L[0] in V[1]):
            return False
          L=L[1:]
        if findvalues(t2[1],unknownassignments)==[]:
             unknownassignments=[[t2[1],t1]]+unknownassignments
             setunknown2(t2[1],t1)
             return True
        return equalt(findvalues(t2[1],unknownassignments)[0],t1)
    if ((not(t1[0]==t2[0]))
        and (t1[0]=="let"
             or t1[0]=="defined"
             or t2[0]=="let" or
             t2[0]=="defined")):
        return equalt(defexpandt(t1),defexpandt(t2))
    if t1[0]=="var":

        return False
    if not(t1[0]==t2[0]):  return False
    if t1[0]=="set":
        v=newvariable(["var",t1[1],t1[2]])
        return equalf(subsf(t1[1],t1[2],v,t1[3]),
                      subsf(t2[1],t2[2],v,t2[3]))
```

```
    if t1[0]=="defined":
        if t1[1]==t2[1]:  return True

        return False
    if t1[0]=="let":
        if not(equalt(t1[1],t2[1])):

            return False
        if not(equalt(t1[2],t2[2])):


            return False
        return equalt(t1[3],t2[3])

# dynamic higher order matching is now installed in equalf
# if one matches with a statement of membership in an unknown
# it will try to set the unknown
# to make formulas equivalent.

def equalf(f1,f2):
    global unknownassignments
    if isrelation(f1[0]) and f1[0]==f2[0]:
        if equalt(f1[1],f2[1]) and equalt(f1[2],f2[2]):  return True
    #print ([f1[0],f1[2][0], findunknown(f1[2])])
    if (f1[0]=="e" and f1[2][0]=="var"
        and not(findunknown(f1[2][1])=="error")):
           #print("Got here")

           F=renamevarf2(f1[1],f1[2],f2)
           print("F is")
           print(F)
           V=findunknown(f1[2][1])
           print("V is")
           print (V)

           T=F
           L = freshvarlistf(T[0])
           print("L is")
           print(L)
           while(not(L==[])):
              if not(L[0] in V[1]):
                 return False
              L=L[1:]
           #print("The term after renaming")
           #print(F)
           S=stratify(graphf(F[0]),F[1])
```

```
        #print(S)
        if S[0]=="stratification failure":
            #print("first return p2][oint")
            return False
        #print(displayt(["set",F[1][1],F[1][2],F[0]]))
        if findvalues(f1[2][1],unknownassignments)==[]:
             unknownassignments=   ([[f1[2][1],
                                      ["set",F[1][1],F[1][2],F[0]]]]
                                    +unknownassignments)
             setunknown2(f1[2][1],["set",F[1][1],F[1][2],F[0]])
             return True
        return equalt(["set",F[1][1],F[1][2],F[0]],
                      findvalues(f1[2][1],
                                 unknownassignments)[0])
#print ([f2[0],f2[2][0], findunknown(f2[2][1])])
if (f2[0]=="e" and f2[2][0]=="var"
    and not(findunknown(f2[2][1])=="error")):
        #print("Got here")


        F=renamevarf2(f2[1],f2[2],f1)
        print("F is")
        print(F)
        V=findunknown(f2[2][1])
        print("V is")
        print(V)
        T=F
        L = freshvarlistf(T[0])
        print("L is")
        print(L)
        while(not(L==[])):
           if not(L[0] in V[1]):
              return False
           L=L[1:]
        #print("The term after renaming")
        #print(F)
        S=stratify(graphf(F[0]),F[1])
        #displayt(["set",F[1][1],F[1][2],F[0]])
        if S[0]=="stratification failure":  return False
        #print(displayt(["set",F[1][1],F[1][2],F[0]]))
        if findvalues(f2[2][1],unknownassignments)==[]:
             unknownassignments=   ([[f2[2][1],
                                      ["set",F[1][1],F[1][2],F[0]]]]
                                    +unknownassignments)
             setunknown2(f2[2][1],["set",F[1][1],F[1][2],F[0]])
             return True
```

```
            return equalt(["set",F[1][1],F[1][2],F[0]],
                          findvalues(f2[2][1],
                                     unknownassignments)[0])
        if (not(f1[0]==f2[0])) and (f1[0]=="let" or f2[0]=="let"):
            return equalf(defexpandf(f1),defexpandf(f2))

        if isrelation(f1[0]) or isrelation(f2[0]):
            #print("second return point")
            return False
        if not(f1[0]==f2[0]):  return False
        if f1[0]=="~":  return equalf(f1[1],f2[1])
        if isconnective(f1[0]):
            if equalf(f1[1],f2[1]) and equalf(f1[2],f2[2]):  return True
            return False
        if isquantifier(f1[0]):
            v=newvariable(["var",f1[1],f1[2]])
            return equalf(subsf(f1[1],f1[2],v,f1[3]),
                          subsf(f2[1],f2[2],v,f2[3]))
        if f1[0]=="let":
            if not(equalt(f1[1],f2[1])):



                return False
            if not(equalt(f1[2],f2[2])):


                return False
            return equalt(f1[3],f2[3]) #equalt is correct here

    # USER COMMAND
    # recognize sequent axioms.  I believe this is the only place
    # where equality of terms and formulas is used so far.
    # Bespoke rules for equality
    # would use it no doubt.

    # because of higher order matching, there are in some cases axioms
    # in proofs where P(a) is identified with a e {x:P(x)}.

    def done():

        global theproof
        global theline
        global unknownassignments
        unknownassignments=[]
        # adding reflexivity of equality as an axiom
```

```
    if len(theproof[theline][1])>0:
        if theproof[theline][1][0][0]=="=":
            if equalt(theproof[theline][1][0][1],
                      theproof[theline][1][0][2]):
                usercommand("done()\n")
                theproof[theline][2]=[]
                displaynextline()
                return("Done")
    # continuing to the usual axioms

    if len(theproof[theline][0])==0 or len(theproof[theline][1])==0:
        print("one of the formulas to be compared not found")
        return "one of the formulas to be compared not found"

    if not(equalf(theproof[theline][0][0],
                  theproof[theline][1][0])):
        print("Not done!")
        return "Not done!"
    usercommand("done()\n")
    theproof[theline][2]=[]
    displaynextline()

# move a desired left or right term to the front of the list

# these had to be written differently
# to let the back() function work through them

# USER COMMAND

def getleft(n):

    global theproof

    if n>= len(theproof[theline][0]):
        displaynextline
        return "the left proposition list is not that long"
    usercommand("getleft("+str(n)+")\n")
    theproof= (theproof[0:theline]
               +[[(theproof[theline][0][n:]
                   +theproof[theline][0][0:n]),
                  theproof[theline][1],
                  theproof[theline][2]]]
               +theproof[theline+1:])
    displaynextline()
```

```
# USER COMMAND

def getright(n):
    global theproof

    if n>= len(theproof[theline][1]):
        displaynextline()
        return "the right proposition list is not that long"
    usercommand("getright("+str(n)+")\n")
    theproof= (theproof[0:theline]
               +[[theproof[theline][0],
                  (theproof[theline][1][n:]
                   +theproof[theline][1][0:n]),
                  theproof[theline][2]]]
               +theproof[theline+1:])
    displaynextline()

# USER COMMANDS

# remove line n from the left or right

def pruneleft(n):

    global theproof

    if n>= len(theproof[theline][0]):
        displaynextline
        return "the left proposition list is not that long"
    usercommand("pruneleft("+str(n)+")\n")
    theproof= (theproof[0:theline]
               +[[(theproof[theline][0][n+1:]
                   +theproof[theline][0][0:n]),
                  theproof[theline][1],
                  theproof[theline][2]]]
               +theproof[theline+1:])
    displaynextline()

# USER COMMAND

def pruneright(n):
    global theproof

    if n>= len(theproof[theline][1]):
        displaynextline()
        return "the right proposition list is not that long"
    usercommand("pruneright("+str(n)+")\n")
```

```
    theproof= (theproof[0:theline]
               +[[theproof[theline][0],
                  (theproof[theline][1][n+1:]
                   +theproof[theline][1][0:n]),
                  theproof[theline][2]]]
               +theproof[theline+1:])
    displaynextline()

r'''
```

2.8 The rules

2.8.1 Specific Sequent Rules: Connectives

We give left rules and right rules (premise and conclusion rules) for the commonly used connectives, excluding converse implication and xor. The notations Γ and Δ represent arbitrary finite sets of propositions. The sentence closest to the turnstile on the right or left is the first sentence in the right or left list in the prover's presentation. The premise on the left is the one which is presented first by the prover when the rule is applied.

The left rule for the biconditional is lazily handled by definitional expansion.

In our experience the rule which it is somewhat difficult to get used to is the left rule for implication, though with a little thought it can be seen to express the rule of *modus ponens*.

2.8.2 Axiom

$$\Gamma, A \vdash A, \Delta$$

2.8.3 Left rules (Premise rules)

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}$$

$$\frac{\Gamma, A \to B, B \to A \vdash \Delta}{\Gamma, A \leftrightarrow B \vdash \Delta}$$

2.8.4 Right rules (Conclusion rules)

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$$

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}$$

$$\frac{\Gamma, A \vdash B, \Delta \qquad \Gamma, B \vdash A, \Delta}{\Gamma \vdash A \leftrightarrow B, \Delta}$$

2.8.5 More Sequent Rules

For conventions on how rules are to be read in general, see the section on sequent rules for connectives above.

In the rules which follow, if ϕ is a formula, $\phi[t/x]$ is taken to represent the result of substituting the term t for the variable x in ϕ.

2.8.6 Rules for Quantifiers

2.8.7 Left Rules (Premise Rules)

$$\frac{\Gamma, \phi[t/x], (\forall x.\phi) \vdash \Delta}{\Gamma, (\forall x.\phi) \vdash \Delta}$$

where t is any term

$$\frac{\Gamma, \phi[a/x] \vdash \Delta}{\Gamma, (\exists x.\phi) \vdash \Delta}$$

where a is a variable not appearing in the conclusion

2.8.8 Right Rules (Conclusion Rules)

$$\frac{\Gamma \vdash \phi[a/x], \Delta}{\Gamma \vdash (\forall x.\phi), \Delta}$$

where a is a variable not appearing in the conclusion

$$\frac{\Gamma \vdash \phi[t/x], (\exists x.\phi), \Delta}{\Gamma \vdash (\exists x.\phi), \Delta}$$

where t is any term

2.8.9 Comments on Quantifier Rules

In some of the quantifier rules, we have retained the quantified sentence from the conclusion in the premise. This is so that we can avoid formalizing notions of copying and reordering formulas in sequents: a quantified formula may be reused several times in a proof, and if it were erased by the application of the rule we would need to copy it explicitly. Another advantage is that it preserves precise equivalence of the conclusion with the conjunction of all the premises, which is a feature of all the sequent rules of Marcel.

The rules requiring input of a new variable a supply a computer-generated variable. In the original version of this prover, the rules involving a new term t were implemented by separate commands with the term t as a parameter. In the current version, the computer supplies a new "unknown variable" which can subsequently be replaced by a term: the advantage is that the same command can then handle all basic sequent rules. The length and readability of computer generated new variables is greatly improved by having two primes instead of one.

2.8.10 Rules for Membership

2.8.11 Left Rule

$$\frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, t \in \{x \mid \phi\} \vdash \Delta}$$

when ϕ is stratified (this condition is handled at parse time)

2.8.12 Right Rule

$$\frac{\Gamma \vdash \phi[t/x], \Delta}{\Gamma \vdash t \in \{x \mid \phi\}, \Delta}$$

when ϕ is stratified (this condition is handled at parse time)

2.8.13 Rules for Equality

2.8.14 Left Rule

$$\frac{\Gamma, (\forall v : t \in v \leftrightarrow u \in v) \vdash \Delta}{\Gamma, t = u \vdash \Delta}$$

This is just definitional expansion. It allows substitution only in stratified contexts, but this is enough for the full strength of substitution as usually presented.

2.8.15 Right Rule

$$\frac{\Gamma \vdash (Ax.x \in t \leftrightarrow x \in u), \Delta}{\Gamma \vdash t = u, \Delta}$$

This gives full extensionality. It can be revised to give SF (use the same Leibniz formulation as in the left rule) or NFU, a bit more elaborate.

To improve the proof theoretic behavior of extensionality, we added a new left rule for membership, tentatively, which expands $x \in y$ to

$$(\forall v : (\forall w : w \in x \leftrightarrow w \in v) \rightarrow v \in y)$$

on the left (when the other rule does not apply). We can report that this allows proof of Leibniz equality from coextensionality without cut. [we revised the exact form of this rule 5/16/2026].

I have also added reflexivity of equality as an axiom. It is provable from the other axioms, but it is convenient not to have to do it over and over.

2.8.16 Global Substitution

The left rule for the universal quantifier and the right rule for the existential quantifier require input of a specific term t. What the prover actually supplies is an fresh variable identified as an "unknown", annotated with the list of all fresh variables generated before it.

The variables introduced by the other quantifier rules we call "arbitrary" variables. When unknown and arbitrary variables are considered together, I call them fresh variables.

The `setunknown` command allows the user to set the unknown variable to a particular term. There is a restriction on what terms can replace an unknown variable: the term must have been definable at the time that the unknown variable was generated, so it cannot contain any free or unknown variables which were not already declared at the time it was introduced (this data is stored with the unknown variable).

It is worth noting that the display function suffixes arbitrary variables with ! and unknown variables with ?, and it forbids variables of the same shape which are bound from being introduced. The parser does not require or even understand these suffixes.

An advantage of this approach are that it enables one to delay the choice of a witness: the later progress of the proof may make it more evident what witness will work.

```
,,,

# leftaction and rightaction are the meat of the logic

def leftaction(f):
    global unknowns
    global newint
    global freshvars
    global variables
    if f[0]=="let":  return [[[defexpandf(f)],[]]]
```

```
    if f[0]=="&":  return [[[f[1],f[2]],[]]]
    if f[0]=="V":  return [[[f[1]],[]],[[f[2]],[]]]
    if f[0]==">":  return [[[],[f[1]]],[[f[2]],[]]]
    if f[0]=="X":
        A=[">",f[1],f[2]]
        B=[">",f[2],f[1]]
        return [[[A,B],[]]]
    if f[0]=="~":  return [[[],[f[1]]]]
    if f[0]=="e" and f[2][0]=="set":
        return [[[subsf(f[2][1],f[2][2],f[1],f[2][3])],[]]]
    if f[0]=="e" and (f[2][0]=="let" or f[2][0]=="defined"):
        return [[[[f[0],f[1],defexpandt(f[2])]],[]]]
    if f[0]=="e" and (f[1][0]=="let" or f[1][0]=="defined"):
        return [[[[f[0],defexpandt(f[1]),f[2]]],[]]]
    if f[0]=="e":
        newint=newint+3
        v=newvariable(["var","x",newint-2])
        w=newvariable(["var","x",newint-1])
        A=["A",v[1],v[2],[">",["A",w[1],w[2],
                               ["X",["e",w,f[1]],
                                ["e",w,v]]],["e",v,f[2]]]]
        return [[[A],[]]]
    #if f[0]=="=" and (f[2][0]=="let" or f[2][0]=="defined"):
        #return [[[[f[0],f[1],defexpandt(f[2])]],[]]]
    #if f[0]=="=" and (f[1][0]=="let" or f[1][0]=="defined"):
        #return [[[[f[0],defexpandt(f[1]),f[2]]],[]]]

    if f[0]=="=":
        newint=newint+1
        v=newvariable(["var","x",newint])
        return [[[["A",v[1],v[2],
                   ["X",["e",f[1],v],
                    ["e",f[2],v]]]],[]]]
    if f[0]=="A":
        v=newvariable(["var",f[1],f[2]])
        unknowns=[[v[1],freshvars]]+unknowns
        freshvars=[v[1]]+freshvars
        return [[[subs1f(f[1],f[2],v,f[3]),f],[]]]
    if f[0]=="E":
        v=newvariable(["var",f[1],f[2]])
        freshvars=[v[1]]+freshvars
        return [[[subs1f(f[1],f[2],v,f[3])],[]]]

    return [[]]

def rightaction(f):
```

```
    global unknowns
    global newint
    global freshvars
    global variables
    if f[0]=="let":  return [[[],[defexpandf(f)]]]
    if f[0]=="&":  return[[[],[f[1]]],[[],[f[2]]]]
    if f[0]=="V":  return [[[],[f[1],f[2]]]]
    if f[0]==">":  return [[[f[1]],[f[2]]]]
    if f[0]=="X":  return [[[f[1]],[f[2]]],[[f[2]],[f[1]]]]
    if f[0]=="~":  return [[[f[1]],[]]]
    if f[0]=="e" and f[2][0]=="set":
        return [[[],[subsf(f[2][1],f[2][2],f[1],f[2][3])]]]
    if f[0]=="e" and (f[2][0]=="let" or f[2][0]=="defined"):
        return [[[],[[f[0],f[1],defexpandt(f[2])]]]]
    if f[0]=="e" and (f[1][0]=="let" or f[1][0]=="defined"):
        return [[[],[[f[0],defexpandt(f[1]),f[2]]]]]



    if f[0]=="=":
        newint=newint+1
        v=newvariable(["var","x",newint])
        return [[[],[["A",v[1],v[2],
                      ["X",["e",v,f[1]],["e",v,f[2]]]]]]]
    if f[0]=="A":
        v=newvariable(["var",f[1],f[2]])
        freshvars=[v[1]]+freshvars
        return [[[],[subs1f(f[1],f[2],v,f[3])]]]
    if f[0]=="E":
        v=newvariable(["var",f[1],f[2]])
        unknowns=[[v[1],freshvars]]+unknowns
        freshvars=[v[1]]+freshvars
        return [[[],[subs1f(f[1],f[2],v,f[3]),f]]]

    return [[]]

# find the unknown with a given name:
# the function is needed because unknowns have attached dependency lists

def findunknown(v):
    global unknowns
    u = unknowns
    while (1==1):
        if u==[]:
            #print("There is no such unknown")
            return "error"
```

```
        if u[0][0]==v:  return u[0]
        u=u[1:]

# lists fresh variables introduced by quantifier rules
# (arbitrary objects or "unknowns") to check fresh variable dependencies

def freshvarlistt(t):

    if t[0]=="var" and t[1] in freshvars:  return [t[1]]
    if t[0]=="var":  return []
    if t[0]=="set":  return freshvarlistf(t[3])
    if t[0]=="defined":  return []
    if t[0]=="let":  return freshvarlistt(t[2])+freshvarlistt(t[3])

def freshvarlistf(f):

    if isrelation(f[0]):  return freshvarlistt(f[1])+freshvarlistt(f[2])
    if f[0]=="~":  return freshvarlistf(f[1])
    if isconnective(f[0]):  return freshvarlistf(f[1])+freshvarlistf(f[2])
    if isquantifier(f[0]):  return freshvarlistf(f[3])
    if f[0]=="let" and f[3][0] == "defined":  return freshvarlistt(f[2])
    if f[0]=="let":  return freshvarlistt(f[2])+freshvarlistf(f[3])

# we use this device to replace witnesses introduced by right
# existential or left universal rules;  in this way we do not have
# to supply them at the time we introduce them.  We do need to
# check that they are something we could have written at the time
# the rule was applied:  this is the reason we have the fresh variables
# introduced prior to an "unknown" as part of the list.

# the idea here is all the quantifier rules act in the same way
# (so they can be crushed into left() and right(), but the introduction
# of a concrete witness can be carried out later -- even some time later
# when the proof suggests what the witness should be.

# I have added the ability to globally replace variables free in the sequent
# with any term which does not contain a fresh variable.  This makes variables
# with free variables usable, though they remain of limited usefulness.
# Universal closures of such theorems are much preferable.

# USER COMMAND

def setunknown(v,t):
    global theproof
    global newint
    global proofstack
```

```
T=gett(t)
if not(isterm(T)):
    print("Bad term entered")
    displaynextline()
    return "Bad term entered"
L = freshvarlistt(T)

V=findunknown(v)
if V == "error":
    if v in freshvars:
        print("Cannot set an arbitrary variable to a value")
        displaynextline()
        return "Cannot set an arbitrary variable to a value"
    if not(L==[]):
        print("Cannot replace with term containing fresh variables")
        displaynextline()
        return "Cannot replace with term containing fresh variables"
    usercommand('setunknown ("'+v+'","'+t+'")\n')

    P=theproof
    P2=[]
    while(not(P==[])):
         P2=P2+[subs1s(v,T,P[0])]
         P=P[1:]
    theproof=P2
    displaynextline()
    return("global free variable handled")


while(not(L==[])):
    if not(L[0] in V[1]):
        print("Fresh variable reference error")
        displaynextline()
        return "Fresh variable reference error"
    L=L[1:]
usercommand('setunknown ("'+v+'","'+t+'")\n')
T2=T

P=theproof
P2=[]
while(not(P==[])):
    P2=P2+[subs1s(V[0],T2,P[0])]
    P=P[1:]
theproof=P2
displaynextline()
```

```
#internal version of setunknown for equality function
# we do not make automatic substitutions for free variables based
# on matching.

def setunknown2(v,t):
    global theproof
    global newint
    V=findunknown(v)
    if V == "error":
        print ("not an unknown")
        displaynextline()
        return "error"
    T=t
    if not(isterm(T)):
        print("Bad term entered")
        displaynextline()
        return "Bad term entered"

    L = freshvarlistt(T)
    while(not(L==[])):
        if not(L[0] in V[1]):
            print("Fresh variable reference error")
            displaynextline()
            return "Fresh variable reference error"
        L=L[1:]

    T2=T

    P=theproof
    P2=[]
    while(not(P==[])):
        P2=P2+[subs1s(V[0],T2,P[0])]
        P=P[1:]
    theproof=P2
    #displaynextline()


# USER COMMAND
# the Cut rule

def Cut(f):
    global theproof
    global linestack

    F=getf(f)
    if not(isformula(F)):
```

```
        print("Bad formula entry")
        displaynextlin()
        return("Bad formula entry")
    usercommand('Cut("'+f+'")\n')
    P=theproof[theline]
    P1=[[F]+P[0],P[1],[-1]]
    P2=[P[0],[F]+P[1],[-1]]
    theproof=theproof+[P1]
    theproof=theproof+[P2]
    theproof[theline][2]=[len(theproof)-2,len(theproof)-1]

    linestack = theproof[theline][2]+linestack

    displaynextline()

# USER COMMAND

def LogTheProof():
    logfile.write('"""')
    logfile.write(displayproof(theproof))
    logfile.write('"""')

theorems = []

# USER COMMAND

def savetheorem(s):
    global theorems

    if not (findvalues(s,theorems)==[]):  return "Theorem "+s+" already exists"
    if not (theline==len(theproof)):  return "Not done!"
    usercommand("savetheorem('"+s+"')\n")
    theorems = [[s,[theproof[0][1][0],theproof]]]+theorems
    print (s+":= "+(displayf(theorems[0][1][0])))

def showtheorem(s):

    if findvalues(s,theorems)==[]:
        print("Theorem "+s+" not found")
        return "Not found"
    print (s+":= "+(displayf(findvalues(s,theorems)[0][0])))

def showtheorems():
    T=rev(theorems)
    while(not T==[]):
        showtheorem(T[0][0])
```

```
        print("Hit return for next theorem")
        input()
        T=T[1:]

def showdefts():
    T=rev(termdefs)
    while(not T==[]):
        showdeft(T[0][0])
        print("Hit return for next term definition")
        input()
        T=T[1:]

def showdeffs():
    T=rev(formuladefs)
    while(not T==[]):
        showdeff(T[0][0])
        print("Hit return for next formula definition")
        input()
        T=T[1:]


# copies of theorems introduced by theoremcut
# are "atified" for good variable management.
# the ability to use setunknown on free variables
# means that theorems with free variables
# can be used.

# USER COMMAND

def theoremcut(s):
    global theproof
    global linestack
    global proofstack


    P=theproof[theline]
    if findvalues(s,theorems)==[]:  return "Theorem "+s+" does not exist"
    usercommand("theoremcut('"+s+"')\n")
    S=raisef(findvalues(s,theorems)[0][0])
    #Q=findvalues(s,theorems)[0][1][0]
    R=[[],[hashifyf(S)],[s]]
    T=[[hashifyf(S)]+P[0],P[1],[-1]]
    theproof=theproof+[R]+[T]
    theproof[theline][2]=[len(theproof)-2,len(theproof)-1]
    linestack=[len(theproof)-1]+linestack
```

```
    displaynextline()

# make sure there is always a log file

# install a variable elimination command.  Use an equation on the left
# to eliminate a variable then drop the equation.
#  If both sides of the equation are variables,
# eliminate the one introduced latest.

# It should be safe to replace the replaceable variable with the other

def replaceable(t,u):
    global freshvars
    if not(t[0])=="var" or not(u[0]=="var"):
        type("Inappropriate use of variable priority function")
        return True
    if not(t[1] in freshvars): return False
    if not(u[1] in freshvars): return True
    U=findunknown(u[1])
    if not(U=="error"):
        if (t[1]in U[1]):  return False
    T=findunknown(t[1])
    if not(T=="error"):
        if not(u[1] in T[1]):  return False
    if U=="error" and T=="error":
        return replaceablearbitrary(t,u)
    return True

# we do not want to make a replacement
# of arbitrary variables which impedes setting an unknown

def replaceablearbitrary(t,u):
    global unknowns
    L=unknowns
    while(not(L==[])):
        if not(t in L[0][1]) and u in L[0][1]:  return True
        if not(u in L[0][1]) and t in L[0][1]:  return False
        L=L[1:]
    return True


# USER COMMAND

#globally eliminate a fresh variable from a sequent
#using the leading equation on the left and removing it
```

```
def varelim():

    global theproof
    global linestack
    global freshvars
    global proofstack
    if len(theproof[theline][0])==0:
        print("No left proposition to eliminate")
        return("error")
    if not(theproof[theline][0][0][0]=="="):
        print("The leading left proposition is not an equation")
        return("error")
    L=theproof[theline][0][0][1]
    R=theproof[theline][0][0][2]
    if (not(L[0] =="var"
        and L[1] in freshvars)
        and not(R[0]=="var" and R[1] in freshvars)):
        print("Neither term in the leading equation is a fresh variable")
    if not(R[0] =="var" and R[1] in freshvars):
        if L[1] in freshvarlistt(R):  return "circularity error"
        V=L[1]
        T=R
    if not(L[0] =="var" and L[1] in freshvars):
        #print(L[1] in freshvars)
        if R[1] in freshvarlistt(L):  return "circularity error"
        V=R[1]
        T=L
    if L[0]=="var" and R[0]=="var" and replaceable(L,R):
        V=L[1]
        T=R
    if L[0]=="var" and R[0]=="var" and not(replaceable(L,R)):
        V=R[1]
        T=L
    usercommand("varelim()\n")
    S=subs1s(V,T,[theproof[theline][0][1:],theproof[theline][1],[-1]])
    theproof[theline][2]=[len(theproof)]
    theproof=theproof+[S]
    linestack=theproof[theline][2]+linestack

    displaynextline()


setlog("autolog")
r'''
```

3 Some sample script files

The "by lines [-1]" annotations have to do with the fact that the lines are not justified at the time they are displayed in a transcript. If I displayed the saved proofs, the list would refer to the numbers of lines justifying the given line, But the displayed proof file would not show the commands used to carry out the proof.

We note here (5/6/2026) that we have just installed the ability to log all user commands to an executable file, so work can be saved and expanded on (and also run again after revisions of the software. All example files are now executable Python files generated by the prover as logs of proofs I carried out. This looks much the same as what I pasted from Python windows in earlier versions, because the logs contain the state of the display at each step in comments.

It can be noted that scripts are *enormous*. They are not as enormous as they appear to be: the text of a script would be much shorter if it were only the command lines, but also very hard to understand. We choose to include the generated output displays, and we have not been economical in terms of space in implementing our output displays. The proofs are long because they are made of many small steps without automation. We could perhaps investigate the implementation of proof tactics in Marcel, but we have not done so so far.

3.1 Inclusion is transitive

Here is a proof that inclusion is transitive. The text was generated by the new scripting features of the system. In this proof, we used the `LogTheProof()` command to append the final form of the proof to the script file as a comment.

```
from graph2 import *

deff ("C", "Ax>exaexb")
start ("AxAyAz > & :ax:byC :ay:bzC :ax:bzC")

"""

Line number 0



----



0.  (Ax : (Ay : (Az : (((x C y) & (y C z)) -> (x C z)))))

by lines [-1]
Next!"""
```

```
right()

"""

Line number 1



----



0.  (Ay : (Az : (((x'! C y) & (y C z)) -> (x'! C z))))

by lines [-1]
Next!"""

right()

"""

Line number 2



----



0.  (Az : (((x'! C y'!) & (y'! C z)) -> (x'! C z)))

by lines [-1]
Next!"""

right()

"""

Line number 3



----
```

```
0.  (((x'! C y'!) & (y'! C z'!)) -> (x'! C z'!))

by lines [-1]
Next!"""

right()

"""

Line number 4



0.  ((x'! C y'!) & (y'! C z'!))
----



0.  (x'! C z'!)

by lines [-1]
Next!"""

right()

"""

Line number 5



0.  ((x'! C y'!) & (y'! C z'!))
----



0.  (A@x : (@x e x'! -> @x e z'!))

by lines [-1]
Next!"""

right()
```

```
"""

Line number 6



0.  ((x'! C y'!) & (y'! C z'!))
----



0.  (x*! e x'! -> x*! e z'!)

by lines [-1]
Next!"""

right()

"""

Line number 7



0.  x*! e x'!

1.  ((x'! C y'!) & (y'! C z'!))
----



0.  x*! e z'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 7
```

```
0.  ((x'! C y'!) & (y'! C z'!))

1.  x*! e x'!
----



0.  x*! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 8



0.  (x'! C y'!)

1.  (y'! C z'!)

2.  x*! e x'!
----



0.  x*! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 9



0.  (A@x : (@x e x'! -> @x e y'!))
```

```
1.  (y'! C z'!)

2.  x*! e x'!
----



0.  x*! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 10



0.  (x''? e x'! -> x''? e y'!)

1.  (A@x : (@x e x'! -> @x e y'!))

2.  (y'! C z'!)

3.  x*! e x'!
----



0.  x*! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 11
```

```
0.  (A@x : (@x e x'! -> @x e y'!))

1.  (y'! C z'!)

2.  x*! e x'!
----



0.  x''? e x'!

1.  x*! e z'!

by lines [-1]
Next!"""

setunknown ("x''","*x")

"""

Line number 11



0.  (A@x : (@x e x'! -> @x e y'!))

1.  (y'! C z'!)

2.  x*! e x'!
----



0.  x*! e x'!

1.  x*! e z'!

by lines [-1]
Next!"""

getleft(2)

"""
```

```
Line number 11



0.  x*! e x'!

1.  (A@x : (@x e x'! -> @x e y'!))

2.  (y'! C z'!)
----



0.  x*! e x'!

1.  x*! e z'!

by lines [-1]
Next!"""

done()

"""

Line number 12



0.  x*! e y'!

1.  (A@x : (@x e x'! -> @x e y'!))

2.  (y'! C z'!)

3.  x*! e x'!
----



0.  x*! e z'!

by lines [-1]
Next!"""
```

```
getleft(2)

"""

Line number 12



0.  (y'! C z'!)

1.  x*! e x'!

2.  x*! e y'!

3.  (A@x : (@x e x'! -> @x e y'!))
----



0.  x*! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 13



0.  (A@x : (@x e y'! -> @x e z'!))

1.  x*! e x'!

2.  x*! e y'!

3.  (A@x : (@x e x'! -> @x e y'!))
----



0.  x*! e z'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 14



0.  (x'*? e y'! -> x'*? e z'!)

1.  (A@x : (@x e y'! -> @x e z'!))

2.  x*! e x'!

3.  x*! e y'!

4.  (A@x : (@x e x'! -> @x e y'!))
----



0.  x*! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 15



0.  (A@x : (@x e y'! -> @x e z'!))

1.  x*! e x'!

2.  x*! e y'!
```

```
3.  (A@x : (@x e x'! -> @x e y'!))
----



0.  x'*? e y'!

1.  x*! e z'!

by lines [-1]
Next!"""

setunknown ("x'*","*x")

"""

Line number 15



0.  (A@x : (@x e y'! -> @x e z'!))

1.  x*! e x'!

2.  x*! e y'!

3.  (A@x : (@x e x'! -> @x e y'!))
----



0.  x*! e y'!

1.  x*! e z'!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 15
```

```
0.  x*! e y'!

1.  (A@x : (@x e x'! -> @x e y'!))

2.  (A@x : (@x e y'! -> @x e z'!))

3.  x*! e x'!
----



0.  x*! e y'!

1.  x*! e z'!

by lines [-1]
Next!"""

done()

"""

Line number 16



0.  x*! e z'!

1.  (A@x : (@x e y'! -> @x e z'!))

2.  x*! e x'!

3.  x*! e y'!

4.  (A@x : (@x e x'! -> @x e y'!))
----



0.  x*! e z'!

by lines [-1]
Next!"""
```

```
done()

"""Done!"""

"""

Line number 0


----


0.  (Ax : (Ay : (Az : (((x C y) & (y C z)) -> (x C z)))))

by lines [1]

Line number 1


----


0.  (Ay : (Az : (((x'! C y) & (y C z)) -> (x'! C z))))

by lines [2]

Line number 2


----


0.  (Az : (((x'! C y'!) & (y'! C z)) -> (x'! C z)))

by lines [3]

Line number 3
```

```
----


0.  (((x'! C y'!) & (y'! C z'!)) -> (x'! C z'!))

by lines [4]

Line number 4



0.  ((x'! C y'!) & (y'! C z'!))
----


0.  (x'! C z'!)

by lines [5]

Line number 5



0.  ((x'! C y'!) & (y'! C z'!))
----


0.  (A@x : (@x e x'! -> @x e z'!))

by lines [6]

Line number 6



0.  ((x'! C y'!) & (y'! C z'!))
----
```

```
0.  (x*! e x'! -> x*! e z'!)

by lines [7]

Line number 7



0.  ((x'! C y'!) & (y'! C z'!))

1.  x*! e x'!
----



0.  x*! e z'!

by lines [8]

Line number 8



0.  (x'! C y'!)

1.  (y'! C z'!)

2.  x*! e x'!
----



0.  x*! e z'!

by lines [9]

Line number 9



0.  (A@x : (@x e x'! -> @x e y'!))
```

```
1.  (y'! C z'!)

2.  x*! e x'!
----



0.  x*! e z'!

by lines [10]

Line number 10



0.  (x*! e x'! -> x*! e y'!)

1.  (A@x : (@x e x'! -> @x e y'!))

2.  (y'! C z'!)

3.  x*! e x'!
----



0.  x*! e z'!

by lines [11, 12]

Line number 11



0.  x*! e x'!

1.  (A@x : (@x e x'! -> @x e y'!))

2.  (y'! C z'!)
----



0.  x*! e x'!
```

```
1.  x*! e z'!

by lines []

Line number 12



0.  (y'! C z'!)

1.  x*! e x'!

2.  x*! e y'!

3.  (A@x : (@x e x'! -> @x e y'!))
----



0.  x*! e z'!

by lines [13]

Line number 13



0.  (A@x : (@x e y'! -> @x e z'!))

1.  x*! e x'!

2.  x*! e y'!

3.  (A@x : (@x e x'! -> @x e y'!))
----



0.  x*! e z'!

by lines [14]

Line number 14
```

```
0.  (x*! e y'! -> x*! e z'!)

1.  (A@x : (@x e y'! -> @x e z'!))

2.  x*! e x'!

3.  x*! e y'!

4.  (A@x : (@x e x'! -> @x e y'!))
----



0.  x*! e z'!

by lines [15, 16]

Line number 15



0.  x*! e y'!

1.  (A@x : (@x e x'! -> @x e y'!))

2.  (A@x : (@x e y'! -> @x e z'!))

3.  x*! e x'!
----



0.  x*! e y'!

1.  x*! e z'!

by lines []

Line number 16
```

```
0.  x*! e z'!

1.  (A@x : (@x e y'! -> @x e z'!))

2.  x*! e x'!

3.  x*! e y'!

4.  (A@x : (@x e x'! -> @x e y'!))
----



0.  x*! e z'!

by lines []"""
```

3.2 Equality is transitive

Here is a proof that equality is transitive, upgraded to use higher order matching.

```
from graph2 import *

start ("AxAyAz>&=xy=yz=xz")

"""

Line number 0



----



0.  (Ax : (Ay : (Az : ((x = y & y = z) -> x = z))))

by lines [-1]
Next!"""

right()

"""
```

```
Line number 1



----



0.  (Ay : (Az : ((x'! = y & y = z) -> x'! = z)))

by lines [-1]
Next!"""

right()

"""

Line number 2



----



0.  (Az : ((x'! = y'! & y'! = z) -> x'! = z))

by lines [-1]
Next!"""

right()

"""

Line number 3



----



0.  ((x'! = y'! & y'! = z'!) -> x'! = z'!)

by lines [-1]
Next!"""
```

```
right()

"""

Line number 4



0.  (x'! = y'! & y'! = z'!)
----



0.  x'! = z'!

by lines [-1]
Next!"""

right()

"""

Line number 5



0.  (x'! = y'! & y'! = z'!)
----



0.  (Ax* : (x* e x'! == x* e z'!))

by lines [-1]
Next!"""

right()

"""

Line number 6
```

```
0.  (x'! = y'! & y'! = z'!)
----



0.  (x''! e x'! == x''! e z'!)

by lines [-1]
Next!"""

right()

"""

Line number 7




0.  x''! e x'!

1.  (x'! = y'! & y'! = z'!)
----



0.  x''! e z'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 7




0.  (x'! = y'! & y'! = z'!)

1.  x''! e x'!
----
```

```
0.  x''! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 9



0.  x'! = y'!

1.  y'! = z'!

2.  x''! e x'!
----



0.  x''! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 10



0.  (Ax'* : (x'! e x'* == y'! e x'*))

1.  y'! = z'!

2.  x''! e x'!
----
```

```
0.  x''! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 11



0.  (x'! e x*'? == y'! e x*'?)

1.  (Ax'* : (x'! e x'* == y'! e x'*))

2.  y'! = z'!

3.  x''! e x'!
----



0.  x''! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 12



0.  (x'! e x*'? -> y'! e x*'?)

1.  (y'! e x*'? -> x'! e x*'?)
```

```
2.  (Ax'* : (x'! e x'* == y'! e x'*))

3.  y'! = z'!

4.  x''! e x'!
----



0.  x''! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 13



0.  (y'! e x*'? -> x'! e x*'?)

1.  (Ax'* : (x'! e x'* == y'! e x'*))

2.  y'! = z'!

3.  x''! e x'!
----



0.  x'! e x*'?

1.  x''! e z'!

by lines [-1]
Next!"""

getleft(3)

"""

Line number 13
```

```
0.  x''! e x'!

1.  (y'! e x*'? -> x'! e x*'?)

2.  (Ax'* : (x'! e x'* == y'! e x'*))

3.  y'! = z'!
----



0.  x'! e x*'?

1.  x''! e z'!

by lines [-1]
Next!"""

done()

"""

Line number 14




0.  y'! e {x** | x''! e x**}

1.  (y'! e {x** | x''! e x**} -> x'! e {x** | x''! e x**})

2.  (Ax'* : (x'! e x'* == y'! e x'*))

3.  y'! = z'!

4.  x''! e x'!
----



0.  x''! e z'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 15



0.  x''! e y'!

1.  (y'! e {x** | x''! e x**} -> x'! e {x** | x''! e x**})

2.  (Ax'* : (x'! e x'* == y'! e x'*))

3.  y'! = z'!

4.  x''! e x'!
----



0.  x''! e z'!

by lines [-1]
Next!"""

getleft(3)

"""

Line number 15



0.  y'! = z'!

1.  x''! e x'!

2.  x''! e y'!

3.  (y'! e {x** | x''! e x**} -> x'! e {x** | x''! e x**})
```

```
4.  (Ax'* : (x'! e x'* == y'! e x'*))
----



0.  x''! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 16



0.  (Ax** : (y'! e x** == z'! e x**))

1.  x''! e x'!

2.  x''! e y'!

3.  (y'! e {x** | x''! e x**} -> x'! e {x** | x''! e x**})

4.  (Ax'* : (x'! e x'* == y'! e x'*))
----



0.  x''! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 17
```

```
0.  (y'! e x'''? == z'! e x'''?)

1.  (Ax** : (y'! e x** == z'! e x**))

2.  x''! e x'!

3.  x''! e y'!

4.  (y'! e {x** | x''! e x**} -> x'! e {x** | x''! e x**})

5.  (Ax'* : (x'! e x'* == y'! e x'*))
----



0.  x''! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 18



0.  (y'! e x'''? -> z'! e x'''?)

1.  (z'! e x'''? -> y'! e x'''?)

2.  (Ax** : (y'! e x** == z'! e x**))

3.  x''! e x'!

4.  x''! e y'!

5.  (y'! e {x** | x''! e x**} -> x'! e {x** | x''! e x**})

6.  (Ax'* : (x'! e x'* == y'! e x'*))
----
```

```
0.  x''! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 19



0.  (z'! e x'''? -> y'! e x'''?)

1.  (Ax** : (y'! e x** == z'! e x**))

2.  x''! e x'!

3.  x''! e y'!

4.  (y'! e {x** | x''! e x**} -> x'! e {x** | x''! e x**})

5.  (Ax'* : (x'! e x'* == y'! e x'*))
----



0.  y'! e x'''?

1.  x''! e z'!

by lines [-1]
Next!"""

getleft(3)

"""

Line number 19
```

```
0.  x''! e y'!

1.  (y'! e {x** | x''! e x**} -> x'! e {x** | x''! e x**})

2.  (Ax'* : (x'! e x'* == y'! e x'*))

3.  (z'! e x'''? -> y'! e x'''?)

4.  (Ax** : (y'! e x** == z'! e x**))

5.  x''! e x'!
----



0.  y'! e x'''?

1.  x''! e z'!

by lines [-1]
Next!"""

done()

"""

Line number 20



0.  z'! e {x''* | x''! e x''*}

1.  (z'! e {x''* | x''! e x''*} -> y'! e {x''* | x''! e x''*})

2.  (Ax** : (y'! e x** == z'! e x**))

3.  x''! e x'!

4.  x''! e y'!

5.  (y'! e {x** | x''! e x**} -> x'! e {x** | x''! e x**})

6.  (Ax'* : (x'! e x'* == y'! e x'*))
----
```

```
0.  x''! e z'!

by lines [-1]
Next!"""

left()

"""

Line number 21



0.  x''! e z'!

1.  (z'! e {x''* | x''! e x''*} -> y'! e {x''* | x''! e x''*})

2.  (Ax** : (y'! e x** == z'! e x**))

3.  x''! e x'!

4.  x''! e y'!

5.  (y'! e {x** | x''! e x**} -> x'! e {x** | x''! e x**})

6.  (Ax'* : (x'! e x'* == y'! e x'*))
----



0.  x''! e z'!

by lines [-1]
Next!"""

done()

"""

Line number 8
```

```
0.  x''! e z'!

1.  (x'! = y'! & y'! = z'!)
----



0.  x''! e x'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 8



0.  (x'! = y'! & y'! = z'!)

1.  x''! e z'!
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 22



0.  x'! = y'!

1.  y'! = z'!
```

```
2.  x''! e z'!
----



0.  x''! e x'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 22



0.  y'! = z'!

1.  x''! e z'!

2.  x'! = y'!
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 23



0.  (Ax''* : (y'! e x''* == z'! e x''*))

1.  x''! e z'!
```

```
2.  x'! = y'!
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 24



0.  (y'! e x'*'? == z'! e x'*'?)

1.  (Ax''* : (y'! e x''* == z'! e x''*))

2.  x''! e z'!

3.  x'! = y'!
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 25



0.  (y'! e x'*'? -> z'! e x'*'?)
```

```
1.  (z'! e x'*'? -> y'! e x'*'?)

2.  (Ax''* : (y'! e x''* == z'! e x''*))

3.  x''! e z'!

4.  x'! = y'!
----



0.  x''! e x'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 25



0.  (z'! e x'*'? -> y'! e x'*'?)

1.  (Ax''* : (y'! e x''* == z'! e x''*))

2.  x''! e z'!

3.  x'! = y'!

4.  (y'! e x'*'? -> z'! e x'*'?)
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 26



0.  (Ax''* : (y'! e x''* == z'! e x''*))

1.  x''! e z'!

2.  x'! = y'!

3.  (y'! e x'*'? -> z'! e x'*'?)
----



0.  z'! e x'*'?

1.  x''! e x'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 26



0.  x''! e z'!

1.  x'! = y'!

2.  (y'! e x'*'? -> z'! e x'*'?)

3.  (Ax''* : (y'! e x''* == z'! e x''*))
----



0.  z'! e x'*'?
```

```
1.  x''! e x'!

by lines [-1]
Next!"""

done()

"""

Line number 27



0.  y'! e {x'** | x''! e x'**}

1.  (Ax''* : (y'! e x''* == z'! e x''*))

2.  x''! e z'!

3.  x'! = y'!

4.  (y'! e {x'** | x''! e x'**} -> z'! e {x'** | x''! e x'**})
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 28



0.  x''! e y'!

1.  (Ax''* : (y'! e x''* == z'! e x''*))
```

```
2.  x''! e z'!

3.  x'! = y'!

4.  (y'! e {x'** | x''! e x'**} -> z'! e {x'** | x''! e x'**})
----



0.  x''! e x'!

by lines [-1]
Next!"""

getleft(3)

"""

Line number 28



0.  x'! = y'!

1.  (y'! e {x'** | x''! e x'**} -> z'! e {x'** | x''! e x'**})

2.  x''! e y'!

3.  (Ax''* : (y'! e x''* == z'! e x''*))

4.  x''! e z'!
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 29
```

```
0.  (Ax'** : (x'! e x'** == y'! e x'**))

1.  (y'! e {x'** | x''! e x'**} -> z'! e {x'** | x''! e x'**})

2.  x''! e y'!

3.  (Ax''* : (y'! e x''* == z'! e x''*))

4.  x''! e z'!
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 30



0.  (x'! e x*''? == y'! e x*''?)

1.  (Ax'** : (x'! e x'** == y'! e x'**))

2.  (y'! e {x'** | x''! e x'**} -> z'! e {x'** | x''! e x'**})

3.  x''! e y'!

4.  (Ax''* : (y'! e x''* == z'! e x''*))

5.  x''! e z'!
----
```

```
0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 31



0.  (x'! e x*''? -> y'! e x*''?)

1.  (y'! e x*''? -> x'! e x*''?)

2.  (Ax'** : (x'! e x'** == y'! e x'**))

3.  (y'! e {x'** | x''! e x'**} -> z'! e {x'** | x''! e x'**})

4.  x''! e y'!

5.  (Ax''* : (y'! e x''* == z'! e x''*))

6.  x''! e z'!
----



0.  x''! e x'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 31



0.  (y'! e x*''? -> x'! e x*''?)
```

```
1.  (Ax'** : (x'! e x'** == y'! e x'**))

2.  (y'! e {x'** | x''! e x'**} -> z'! e {x'** | x''! e x'**})

3.  x''! e y'!

4.  (Ax''* : (y'! e x''* == z'! e x''*))

5.  x''! e z'!

6.  (x'! e x*''? -> y'! e x*''?)
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 32



0.  (Ax'** : (x'! e x'** == y'! e x'**))

1.  (y'! e {x'** | x''! e x'**} -> z'! e {x'** | x''! e x'**})

2.  x''! e y'!

3.  (Ax''* : (y'! e x''* == z'! e x''*))

4.  x''! e z'!

5.  (x'! e x*''? -> y'! e x*''?)
----



0.  y'! e x*''?
```

```
1.  x’’! e x’!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 32



0.  x’’! e y’!

1.  (Ax’’* : (y’! e x’’* == z’! e x’’*))

2.  x’’! e z’!

3.  (x’! e x*’’? -> y’! e x*’’?)

4.  (Ax’** : (x’! e x’** == y’! e x’**))

5.  (y’! e {x’** | x’’! e x’**} -> z’! e {x’** | x’’! e x’**})
----



0.  y’! e x*’’?

1.  x’’! e x’!

by lines [-1]
Next!"""

done()

"""

Line number 33
```

```
0.  x'! e {x*'* | x''! e x*'*}

1.  (Ax'** : (x'! e x'** == y'! e x'**))

2.  (y'! e {x'** | x''! e x'**} -> z'! e {x'** | x''! e x'**})

3.  x''! e y'!

4.  (Ax''* : (y'! e x''* == z'! e x''*))

5.  x''! e z'!

6.  (x'! e {x*'* | x''! e x*'*} -> y'! e {x*'* | x''! e x*'*})
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 34



0.  x''! e x'!

1.  (Ax'** : (x'! e x'** == y'! e x'**))

2.  (y'! e {x'** | x''! e x'**} -> z'! e {x'** | x''! e x'**})

3.  x''! e y'!

4.  (Ax''* : (y'! e x''* == z'! e x''*))

5.  x''! e z'!

6.  (x'! e {x*'* | x''! e x*'*} -> y'! e {x*'* | x''! e x*'*})
----
```

```
0.  x''! e x'!

by lines [-1]
Next!"""

done()

"""Done!"""
```

3.3 Coextensionality implies Leibniz equality

This example is perhaps a bit strange. This is a proof that coextensionality implies Leibniz equality: it uses the weird left rule for membership statements which I have introduced experimentally, in order to avoid the need to cut. It is also a nice sequent proof, and examplifies the notation and style of use of the prover, so here it is.

This new version uses the new higher order matching facility.

```
from graph2 import *

start ("AaAb>AxXexaexbAuXeauebu")

"""

Line number 0



----



0.  (Aa : (Ab : ((Ax : (x e a == x e b)) -> (Au : (a e u == b e u)))))

by lines [-1]
Next!"""

right()

"""

Line number 1
```

```
----



0.  (Ab : ((Ax : (x e a'! == x e b)) -> (Au : (a'! e u == b e u))))

by lines [-1]
Next!"""

right()

"""

Line number 2



----



0.  ((Ax : (x e a'! == x e b'!)) -> (Au : (a'! e u == b'! e u)))

by lines [-1]
Next!"""

right()

"""

Line number 3




0.  (Ax : (x e a'! == x e b'!))
----



0.  (Au : (a'! e u == b'! e u))

by lines [-1]
Next!"""
```

```
right()

"""

Line number 4



0.  (Ax : (x e a'! == x e b'!))
----



0.  (a'! e u'! == b'! e u'!)

by lines [-1]
Next!"""

right()

"""

Line number 5



0.  a'! e u'!

1.  (Ax : (x e a'! == x e b'!))
----



0.  b'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 7
```

```
0.  (Ax' : ((Ax* : (x* e a'! == x* e x')) -> x' e u'!))

1.  (Ax : (x e a'! == x e b'!))
----



0.  b'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 8



0.  ((Ax* : (x* e a'! == x* e x''?)) -> x''? e u'!)

1.  (Ax' : ((Ax* : (x* e a'! == x* e x')) -> x' e u'!))

2.  (Ax : (x e a'! == x e b'!))
----



0.  b'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 9
```

```
0.  (Ax' : ((Ax* : (x* e a'! == x* e x')) -> x' e u'!))

1.  (Ax : (x e a'! == x e b'!))
----



0.  (Ax* : (x* e a'! == x* e x''?))

1.  b'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 11




0.  ((Ax* : (x* e a'! == x* e x'*?)) -> x'*? e u'!)

1.  (Ax' : ((Ax* : (x* e a'! == x* e x')) -> x' e u'!))

2.  (Ax : (x e a'! == x e b'!))
----



0.  (Ax* : (x* e a'! == x* e x''?))

1.  b'! e u'!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 11
```

```
0.  (Ax : (x e a'! == x e b'!))

1.  ((Ax* : (x* e a'! == x* e x'*?)) -> x'*? e u'!)

2.  (Ax' : ((Ax* : (x* e a'! == x* e x')) -> x' e u'!))
----



0.  (Ax* : (x* e a'! == x* e x''?))

1.  b'! e u'!

by lines [-1]
Next!"""

done()

"""

Line number 10



0.  b'! e u'!

1.  (Ax' : ((Ax* : (x* e a'! == x* e x')) -> x' e u'!))

2.  (Ax : (x e a'! == x e b'!))
----



0.  b'! e u'!

by lines [-1]
Next!"""

done()

"""
```

```
Line number 6



0.  b'! e u'!

1.  (Ax : (x e a'! == x e b'!))
----



0.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 12



0.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

1.  (Ax : (x e a'! == x e b'!))
----



0.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 13
```

```
0.  ((Ax''' : (x''' e b'! == x''' e x''*?)) -> x''*? e u'!)

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

2.  (Ax : (x e a'! == x e b'!))
----



0.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 14



0.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

1.  (Ax : (x e a'! == x e b'!))
----



0.  (Ax''' : (x''' e b'! == x''' e x''*?))

1.  a'! e u'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 14
```

```
0.  (Ax : (x e a'! == x e b'!))

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  (Ax''' : (x''' e b'! == x''' e x''*?))

1.  a'! e u'!

by lines [-1]
Next!"""

right()

"""

Line number 16



0.  (Ax : (x e a'! == x e b'!))

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  (x'**! e b'! == x'**! e x''*?)

1.  a'! e u'!

by lines [-1]
Next!"""

right()

"""

Line number 17
```

```
0.  x'**! e b'!

1.  (Ax : (x e a'! == x e b'!))

2.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  x'**! e x''*?

1.  a'! e u'!

by lines [-1]
Next!"""

done()

"""

Line number 18



0.  x'**! e b'!

1.  (Ax : (x e a'! == x e b'!))

2.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  x'**! e b'!

1.  a'! e u'!

by lines [-1]
Next!"""

done()

"""
```

```
Line number 15



0.  b'! e u'!

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

2.  (Ax : (x e a'! == x e b'!))
----



0.  a'! e u'!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 15



0.  (Ax : (x e a'! == x e b'!))

1.  b'! e u'!

2.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  a'! e u'!

by lines [-1]
Next!"""

left()

"""
```

```
Line number 19



0.  (x*''? e a'! == x*''? e b'!)

1.  (Ax : (x e a'! == x e b'!))

2.  b'! e u'!

3.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 20



0.  (x*''? e a'! -> x*''? e b'!)

1.  (x*''? e b'! -> x*''? e a'!)

2.  (Ax : (x e a'! == x e b'!))

3.  b'! e u'!

4.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  a'! e u'!
```

```
by lines [-1]
Next!"""

getleft(1)

"""

Line number 20



0.  (x*''? e b'! -> x*''? e a'!)

1.  (Ax : (x e a'! == x e b'!))

2.  b'! e u'!

3.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

4.  (x*''? e a'! -> x*''? e b'!)
----



0.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 21



0.  (Ax : (x e a'! == x e b'!))

1.  b'! e u'!

2.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

3.  (x*''? e a'! -> x*''? e b'!)
```

```
----



0.  x*''? e b'!

1.  a'! e u'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 21



0.  b'! e u'!

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

2.  (x*''? e a'! -> x*''? e b'!)

3.  (Ax : (x e a'! == x e b'!))
----



0.  x*''? e b'!

1.  a'! e u'!

by lines [-1]
Next!"""

back()
back()
back()
back()
back()
back()
getleft(2)
```

```
"""

Line number 15



0.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

1.  (Ax : (x e a'! == x e b'!))

2.  b'! e u'!
----



0.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 19



0.  ((Ax''' : (x''' e b'! == x''' e x*''?)) -> x*''? e u'!)

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

2.  (Ax : (x e a'! == x e b'!))

3.  b'! e u'!
----



0.  a'! e u'!

by lines [-1]
Next!"""
```

```
left()

"""

Line number 20



0.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

1.  (Ax : (x e a'! == x e b'!))

2.  b'! e u'!
----



0.  (Ax''' : (x''' e b'! == x''' e x*''?))

1.  a'! e u'!

by lines [-1]
Next!"""

right()

"""

Line number 22



0.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

1.  (Ax : (x e a'! == x e b'!))

2.  b'! e u'!
----



0.  (x*'*! e b'! == x*'*! e x*''?)

1.  a'! e u'!
```

```
by lines [-1]
Next!"""

right()

"""

Line number 23



0.  x*'*! e b'!

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

2.  (Ax : (x e a'! == x e b'!))

3.  b'! e u'!
----



0.  x*'*! e x*''?

1.  a'! e u'!

by lines [-1]
Next!"""

setunknown ("x*''","'a")

"""

Line number 23



0.  x*'*! e b'!

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

2.  (Ax : (x e a'! == x e b'!))
```

```
3.  b'! e u'!
----



0.  x*'*! e a'!

1.  a'! e u'!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 23




0.  (Ax : (x e a'! == x e b'!))

1.  b'! e u'!

2.  x*'*! e b'!

3.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  x*'*! e a'!

1.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 25
```

```
0.  (x**'? e a'! == x**'? e b'!)

1.  (Ax : (x e a'! == x e b'!))

2.  b'! e u'!

3.  x*'*! e b'!

4.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  x*'*! e a'!

1.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 26



0.  (x**'? e a'! -> x**'? e b'!)

1.  (x**'? e b'! -> x**'? e a'!)

2.  (Ax : (x e a'! == x e b'!))

3.  b'! e u'!

4.  x*'*! e b'!

5.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----
```

```
0.  x*'*! e a'!

1.  a'! e u'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 26



0.  (x**'? e b'! -> x**'? e a'!)

1.  (Ax : (x e a'! == x e b'!))

2.  b'! e u'!

3.  x*'*! e b'!

4.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

5.  (x**'? e a'! -> x**'? e b'!)
----



0.  x*'*! e a'!

1.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 27
```

```
0.  (Ax : (x e a'! == x e b'!))

1.  b'! e u'!

2.  x*'*! e b'!

3.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

4.  (x**'? e a'! -> x**'? e b'!)
----



0.  x**'? e b'!

1.  x*'*! e a'!

2.  a'! e u'!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 27



0.  x*'*! e b'!

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

2.  (x**'? e a'! -> x**'? e b'!)

3.  (Ax : (x e a'! == x e b'!))

4.  b'! e u'!
----



0.  x**'? e b'!
```

```
1.  x*'*! e a'!

2.  a'! e u'!

by lines [-1]
Next!"""

done()

"""

Line number 28




0.  x*'*! e a'!

1.  (Ax : (x e a'! == x e b'!))

2.  b'! e u'!

3.  x*'*! e b'!

4.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

5.  (x*'*! e a'! -> x*'*! e b'!)
----



0.  x*'*! e a'!

1.  a'! e u'!

by lines [-1]
Next!"""

done()

"""

Line number 24
```

```
0.  x*'*! e a'!

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

2.  (Ax : (x e a'! == x e b'!))

3.  b'! e u'!
----



0.  x*'*! e b'!

1.  a'! e u'!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 24



0.  (Ax : (x e a'! == x e b'!))

1.  b'! e u'!

2.  x*'*! e a'!

3.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  x*'*! e b'!

1.  a'! e u'!

by lines [-1]
Next!"""
```

```
left()

"""

Line number 29



0.  (x***? e a'! == x***? e b'!)

1.  (Ax : (x e a'! == x e b'!))

2.  b'! e u'!

3.  x*'*! e a'!

4.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  x*'*! e b'!

1.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 30



0.  (x***? e a'! -> x***? e b'!)

1.  (x***? e b'! -> x***? e a'!)

2.  (Ax : (x e a'! == x e b'!))

3.  b'! e u'!
```

```
4.  x*'*! e a'!

5.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  x*'*! e b'!

1.  a'! e u'!

by lines [-1]
Next!"""

left()

"""

Line number 31



0.  (x***? e b'! -> x***? e a'!)

1.  (Ax : (x e a'! == x e b'!))

2.  b'! e u'!

3.  x*'*! e a'!

4.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  x***? e a'!

1.  x*'*! e b'!

2.  a'! e u'!

by lines [-1]
Next!"""
```

```
getleft(3)

"""

Line number 31



0.  x*'*! e a'!

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

2.  (x***? e b'! -> x***? e a'!)

3.  (Ax : (x e a'! == x e b'!))

4.  b'! e u'!
----



0.  x***? e a'!

1.  x*'*! e b'!

2.  a'! e u'!

by lines [-1]
Next!"""

done()

"""

Line number 32



0.  x*'*! e b'!

1.  (x*'*! e b'! -> x*'*! e a'!)

2.  (Ax : (x e a'! == x e b'!))
```

```
3.  b'! e u'!

4.  x*'*! e a'!

5.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))
----



0.  x*'*! e b'!

1.  a'! e u'!

by lines [-1]
Next!"""

done()

"""

Line number 21



0.  a'! e u'!

1.  (Ax** : ((Ax''' : (x''' e b'! == x''' e x**)) -> x** e u'!))

2.  (Ax : (x e a'! == x e b'!))

3.  b'! e u'!
----



0.  a'! e u'!

by lines [-1]
Next!"""

done()

"""Done!"""
```

3.4 A weird example with unknown variables

The entry line illustrates the new capability to use spaces, parentheses, brackets, and commas as padding in term and formula input. There is no balance checking: parentheses and brackets are just for the person entering the term [to make sure they do it right].

```
from graph2 import *

start ("AyEuAx>exyeuy")

"""

Line number 0



----



0.  (Ay : (Eu : (Ax : (x e y -> u e y))))

by lines [-1]
Next!"""

right()

"""

Line number 1



----



0.  (Eu : (Ax : (x e y'! -> u e y'!)))

by lines [-1]
Next!"""

right()

"""
```

```
Line number 2



----



0.  (Ax : (x e y' -> u'? e y'))

1.  (Eu : (Ax : (x e y' -> u e y')))

by lines [-1]
Next!"""

right()

"""

Line number 3



----



0.  (x'! e y' -> u'? e y')

1.  (Eu : (Ax : (x e y' -> u e y')))

by lines [-1]
Next!"""

right()

"""

Line number 4



0.  x'! e y'
----
```

```
0.  u'? e y'

1.  (Eu : (Ax : (x e y' -> u e y')))

by lines [-1]
Next!"""

done()
getright(1)

"""

Line number 4



0.  x'! e y'
----



0.  (Eu : (Ax : (x e y' -> u e y')))

1.  u'? e y'

by lines [-1]
Next!"""

right()

"""

Line number 5



0.  x' e y'
----



0.  (Ax : (x e y' -> u*? e y'))
```

```
1.  (Eu : (Ax : (x e y' -> u e y')))

2.  u'? e y'

by lines [-1]
Next!"""

right()

"""

Line number 6



0.  x' e y'
----



0.  (x*! e y' -> u*? e y')

1.  (Eu : (Ax : (x e y' -> u e y')))

2.  u'? e y'

by lines [-1]
Next!"""

right()

"""

Line number 7



0.  x*! e y'

1.  x' e y'
----
```

```
0.  u*? e y'

1.  (Eu : (Ax : (x e y' -> u e y')))

2.  u'? e y'

by lines [-1]
Next!"""

getleft(1)

"""

Line number 7




0.  x' e y'

1.  x*! e y'
----



0.  u*? e y'

1.  (Eu : (Ax : (x e y' -> u e y')))

2.  u'? e y'

by lines [-1]
Next!"""

done()

"""Done!"""

"""

Line number 0



----
```

```
0.  (Ay : (Eu : (Ax : (x e y -> u e y))))

by lines [1]

Line number 1


----


0.  (Eu : (Ax : (x e y' -> u e y')))

by lines [2]

Line number 2


----


0.  (Ax : (x e y' -> u'? e y'))

1.  (Eu : (Ax : (x e y' -> u e y')))

by lines [3]

Line number 3


----


0.  (x' e y' -> u'? e y')

1.  (Eu : (Ax : (x e y' -> u e y')))

by lines [4]
```

```
Line number 4



0.  x' e y'
----



0.  (Eu : (Ax : (x e y' -> u e y')))

1.  u'? e y'

by lines [5]

Line number 5



0.  x' e y'
----



0.  (Ax : (x e y' -> x' e y'))

1.  (Eu : (Ax : (x e y' -> u e y')))

2.  u'? e y'

by lines [6]

Line number 6



0.  x' e y'
----



0.  (x*! e y' -> x' e y')
```

```
1.  (Eu : (Ax : (x e y' -> u e y')))

2.  u'? e y'

by lines [7]

Line number 7




0.  x' e y'

1.  x*! e y'
----



0.  x' e y'

1.  (Eu : (Ax : (x e y' -> u e y')))

2.  u'? e y'

by lines []"""
```

3.5 Atomic inclusion is membership

This proves the equivalence of $\{x\} \subseteq y$ and $x \in y$. It involves both term and formula definition.

```
from graph2 import *

deff ("C", "Ax>exaexb")
deft ("I", "{x=xa")
start ("AxAyX:a:axI:byCexy")

"""

Line number 0



----
```

```
0.  (Ax : (Ay : ((I(a:x) C y) == x e y)))

by lines [-1]
Next!"""

right()

"""

Line number 1



----



0.  (Ay : ((I(a:x'!) C y) == x'! e y))

by lines [-1]
Next!"""

right()

"""

Line number 2



----



0.  ((I(a:x'!) C y'!) == x'! e y'!)

by lines [-1]
Next!"""

right()

"""

Line number 3
```

```
0.  (I(a:x'!) C y'!)
----



0.  x'! e y'!

by lines [-1]
Next!"""

left()

"""

Line number 5



0.  (A@x : (@x e I(a:x'!) -> @x e y'!))
----



0.  x'! e y'!

by lines [-1]
Next!"""

left()

"""

Line number 6



0.  (x*? e I(a:x'!) -> x*? e y'!)

1.  (A@x : (@x e I(a:x'!) -> @x e y'!))
----
```

```
0.  x'! e y'!

by lines [-1]
Next!"""

left()

"""

Line number 7



0.  (A@x : (@x e I(a:x'!) -> @x e y'!))
----



0.  x*? e I(a:x'!)

1.  x'! e y'!

by lines [-1]
Next!"""

right()

"""

Line number 9



0.  (A@x : (@x e I(a:x'!) -> @x e y'!))
----



0.  x*? e {@x | @x = x'!}

1.  x'! e y'!
```

```
by lines [-1]
Next!"""

right()

"""

Line number 10



0.  (A@x : (@x e I(a:x'!) -> @x e y'!))
----



0.  x*? = x'!

1.  x'! e y'!

by lines [-1]
Next!"""

right()

"""

Line number 11



0.  (A@x : (@x e I(a:x'!) -> @x e y'!))
----



0.  (Ax'' : (x'' e x*? == x'' e x'!))

1.  x'! e y'!

by lines [-1]
Next!"""
```

```
right()

"""

Line number 12



0.  (A@x : (@x e I(a:x'!) -> @x e y'!))
----



0.  (x'*! e x*? == x'*! e x'!)

1.  x'! e y'!

by lines [-1]
Next!"""

right()

"""

Line number 13



0.  x'*! e x*?

1.  (A@x : (@x e I(a:x'!) -> @x e y'!))
----



0.  x'*! e x'!

1.  x'! e y'!

by lines [-1]
Next!"""

done()
```

```
"""

Line number 14




0.  x'*! e x'!

1.  (A@x : (@x e I(a:x'!) -> @x e y'!))
----



0.  x'*! e x'!

1.  x'! e y'!

by lines [-1]
Next!"""

done()

"""

Line number 8




0.  x'! e y'!

1.  (A@x : (@x e I(a:x'!) -> @x e y'!))
----



0.  x'! e y'!

by lines [-1]
Next!"""

done()

"""
```

```
Line number 4



0.  x'! e y'!
----



0.  (I(a:x'!) C y'!)

by lines [-1]
Next!"""

right()

"""

Line number 15



0.  x'! e y'!
----



0.  (A@x : (@x e I(a:x'!) -> @x e y'!))

by lines [-1]
Next!"""

right()

"""

Line number 16



0.  x'! e y'!
----
```

```
0.  (x*'! e I(a:x'!) -> x*'! e y'!)

by lines [-1]
Next!"""

right()

"""

Line number 17




0.  x*'! e I(a:x'!)

1.  x'! e y'!
----



0.  x*'! e y'!

by lines [-1]
Next!"""

left()

"""

Line number 18




0.  x*'! e {@x | @x = x'!}

1.  x'! e y'!
----



0.  x*'! e y'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 19



0.  x*'! = x'!

1.  x'! e y'!
----



0.  x*'! e y'!

by lines [-1]
Next!"""

left()

"""

Line number 20



0.  (Ax** : (x*'! e x** == x'! e x**))

1.  x'! e y'!
----



0.  x*'! e y'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 21



0.  (x*'! e x'''? == x'! e x'''?)

1.  (Ax** : (x*'! e x** == x'! e x**))

2.  x'! e y'!
----



0.  x*'! e y'!

by lines [-1]
Next!"""

left()

"""

Line number 22



0.  (x*'! e x'''? -> x'! e x'''?)

1.  (x'! e x'''? -> x*'! e x'''?)

2.  (Ax** : (x*'! e x** == x'! e x**))

3.  x'! e y'!
----



0.  x*'! e y'!

by lines [-1]
Next!"""
```

```
getleft(1)

"""

Line number 22



0.  (x'! e x'''? -> x*'! e x'''?)

1.  (Ax** : (x*'! e x** == x'! e x**))

2.  x'! e y'!

3.  (x*'! e x'''? -> x'! e x'''?)
----



0.  x*'! e y'!

by lines [-1]
Next!"""

left()

"""

Line number 23



0.  (Ax** : (x*'! e x** == x'! e x**))

1.  x'! e y'!

2.  (x*'! e x'''? -> x'! e x'''?)
----



0.  x'! e x'''?
```

```
1.  x*'! e y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 23




0.  x'! e y'!

1.  (x*'! e x'''? -> x'! e x'''?)

2.  (Ax** : (x*'! e x** == x'! e x**))
----



0.  x'! e x'''?

1.  x*'! e y'!

by lines [-1]
Next!"""

done()

"""

Line number 24




0.  x*'! e y'!

1.  (Ax** : (x*'! e x** == x'! e x**))

2.  x'! e y'!

3.  (x*'! e y'! -> x'! e y'!)
```

```
----



0.  x*'! e y'!

by lines [-1]
Next!"""

done()

"""Done!"""
```

3.6 Projection theorems for the ordered pair

There may be some wheel spinning in this proof. It contains definitions of the unordered pair, the ordered pair, and the first projection operator on ordered pairs, and the proof that the first projection operator works. This is an enormous text.

```
from graph2 import *

#setlog("newpairslog")
deft ("P", "{xV=xa=xb")
deft ("O", ":a:aa:baP:b:aa:bbPP")
deft ("'P", "{xAy>eyaexy")
start ("AxAy=:a:ax:byO'P{u=xu")

"""

Line number 0



----



0.  (Ax : (Ay : P'(a:(x O y))
   = {u | x = u}))

by lines [-1]
Next!"""

right()
```

```
"""

Line number 1



----



0.  (Ay : P'(a:(x'! O y))
   = {u | x'! = u})

by lines [-1]
Next!"""

right()

"""

Line number 2



----



0.  P'(a:(x'! O y'!))
   = {u | x'! = u}

by lines [-1]
Next!"""

right()

"""

Line number 3



----
```

```
0.  (Ax* : (x* e P'(a:(x'! O y'!))
   == x* e {u | x'! = u}))

by lines [-1]
Next!"""

right()

"""

Line number 4



----



0.  (x''! e P'(a:(x'! O y'!))
   == x''! e {u | x'! = u})

by lines [-1]
Next!"""

right()

"""

Line number 5



0.  x''! e P'(a:(x'! O y'!))
----



0.  x''! e {u | x'! = u}

by lines [-1]
Next!"""

right()
```

```
"""

Line number 7




0.  x''! e P'(a:(x'! O y'!))
----



0.  x'! = x''!

by lines [-1]
Next!"""

left()

"""

Line number 8




0.  x''! e {@x | (A@y : (@y e (x'! O y'!)
   -> @x e @y))}
----



0.  x'! = x''!

by lines [-1]
Next!"""

left()

"""

Line number 9
```

```
0.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----



0.  x'! = x''!

by lines [-1]
Next!"""

left()

"""

Line number 10



0.  (y*? e (x'! O y'!)
   -> x''! e y*?)

1.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----



0.  x'! = x''!

by lines [-1]
Next!"""

left()

"""

Line number 11



0.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----
```

```
0.  y*? e (x'! O y'!)

1.  x'! = x''!

by lines [-1]
Next!"""

right()

"""

Line number 13




0.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----



0.  y*? e ((x'! P x'!)
   P (x'! P y'!))

1.  x'! = x''!

by lines [-1]
Next!"""

right()

"""

Line number 14




0.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----
```

```
0.  y*? e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  x'! = x''!

by lines [-1]
Next!"""

right()

"""

Line number 15




0.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----



0.  (y*? = (x'! P x'!)
   V y*? = (x'! P y'!))

1.  x'! = x''!

by lines [-1]
Next!"""

right()

"""

Line number 16




0.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----
```

```
0.  y*? = (x'! P x'!)

1.  y*? = (x'! P y'!)

2.  x'! = x''!

by lines [-1]
Next!"""

done()

"""

Line number 12




0.  x''! e (x'! P x'!)

1.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----



0.  x'! = x''!

by lines [-1]
Next!"""

left()

"""

Line number 17




0.  x''! e {@x | (@x = x'! V @x = x'!)}

1.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
```

```
----



0.  x'! = x''!

by lines [-1]
Next!"""

left()

"""

Line number 18



0.  (x''! = x'! V x''! = x'!)

1.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----



0.  x'! = x''!

by lines [-1]
Next!"""

left()

"""

Line number 19



0.  x''! = x'!

1.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----
```

```
0.  x'! = x''!

by lines [-1]
Next!"""

varelim()

"""

Line number 21



0.  (A@y : (@y e (x'! O y'!)
   -> x'! e @y))
----



0.  x'! = x'!

by lines [-1]
Next!"""

done()

"""

Line number 20



0.  x''! = x'!

1.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))
----



0.  x'! = x''!
```

```
by lines [-1]
Next!"""

varelim()

"""

Line number 22



0.  (A@y : (@y e (x'! O y'!)
   -> x'! e @y))
----



0.  x'! = x'!

by lines [-1]
Next!"""

done()

"""

Line number 6



0.  x''! e {u | x'! = u}
----



0.  x''! e P'(a:(x'! O y'!))

by lines [-1]
Next!"""

left()

"""
```

```
Line number 23



0.  x'! = x''!
----



0.  x''! e P'(a:(x'! O y'!))

by lines [-1]
Next!"""

right()

"""

Line number 24



0.  x'! = x''!
----



0.  x''! e {@x | (A@y : (@y e (x'! O y'!)
   -> @x e @y))}

by lines [-1]
Next!"""

right()

"""

Line number 25



0.  x'! = x''!
----
```

```
0.  (A@y : (@y e (x'! O y'!)
   -> x''! e @y))

by lines [-1]
Next!"""

right()

"""

Line number 26



0.  x'! = x''!
----



0.  (y''! e (x'! O y'!)
   -> x''! e y''!)

by lines [-1]
Next!"""

varelim()

"""

Line number 27



----



0.  (y''! e (x''! O y'!)
   -> x''! e y''!)

by lines [-1]
Next!"""
```

```
right()

"""

Line number 28



0.  y''! e (x''! O y'!)
----



0.  x''! e y''!

by lines [-1]
Next!"""

left()

"""

Line number 29



0.  y''! e ((x''! P x''!)
   P (x''! P y'!))
----



0.  x''! e y''!

by lines [-1]
Next!"""

left()

"""

Line number 30
```

```
0.  y''! e {@x | (@x = (x''! P x''!)
   V @x = (x''! P y'!))}
----



0.  x''! e y''!

by lines [-1]
Next!"""

left()

"""

Line number 31



0.  (y''! = (x''! P x''!)
   V y''! = (x''! P y'!))
----



0.  x''! e y''!

by lines [-1]
Next!"""

left()

"""

Line number 32



0.  y''! = (x''! P x''!)
----
```

```
0.  x''! e y''!

by lines [-1]
Next!"""

varelim()

"""

Line number 34



----



0.  x''! e (x''! P x''!)

by lines [-1]
Next!"""

right()

"""

Line number 35



----



0.  x''! e {@x | (@x = x''! V @x = x''!)}

by lines [-1]
Next!"""

right()

"""

Line number 36
```

```
----



0.  (x''! = x''! V x''! = x''!)

by lines [-1]
Next!"""

right()

"""

Line number 37



----



0.  x''! = x''!

1.  x''! = x''!

by lines [-1]
Next!"""

done()

"""

Line number 33



0.  y''! = (x''! P y'!)
----



0.  x''! e y''!
```

```
by lines [-1]
Next!"""

varelim()

"""

Line number 38



----



0.  x''! e (x''! P y'!)

by lines [-1]
Next!"""

right()

"""

Line number 39



----



0.  x''! e {@x | (@x = x''! V @x = y'!)}

by lines [-1]
Next!"""

right()

"""

Line number 40
```

```
----



0.  (x''! = x''! V x''! = y'!)

by lines [-1]
Next!"""

right()

"""

Line number 41



----



0.  x''! = x''!

1.  x''! = y'!

by lines [-1]
Next!"""

done()

"""Done!"""

savetheorem("Proj1")

deft ("U", "{xEyAzXezx=zy")
deft ("*P", "{xe{y&exyeyaU")
start ("AxAy=:a:ax:byO*P{u=uy")

"""

Line number 0



----
```

```
0.  (Ax : (Ay : P*(a:(x O y))
   = {u | u = y}))

by lines [-1]
Next!"""

right()

"""

Line number 1



----



0.  (Ay : P*(a:(x'! O y))
   = {u | u = y})

by lines [-1]
Next!"""

right()

"""

Line number 2



----



0.  P*(a:(x'! O y'!))
   = {u | u = y'!}

by lines [-1]
Next!"""

right()
```

```
"""

Line number 3



----



0.  (Ax* : (x* e P*(a:(x'! O y'!))
   == x* e {u | u = y'!}))

by lines [-1]
Next!"""

right()

"""

Line number 4



----



0.  (x''! e P*(a:(x'! O y'!))
   == x''! e {u | u = y'!})

by lines [-1]
Next!"""

right()

"""

Line number 5



0.  x''! e P*(a:(x'! O y'!))
----
```

```
0.  x''! e {u | u = y'!}

by lines [-1]
Next!"""

right()

"""

Line number 7




0.  x''! e P*(a:(x'! O y'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 8




0.  x''! e {@x | {@y | (@x e @y
   & @y e (x'! O y'!))} e U}
----



0.  x''! = y'!

by lines [-1]
Next!"""
```

```
left()

"""

Line number 9



0.  {@y | (x''! e @y
   & @y e (x'! O y'!))} e U
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 10



0.  {@y | (x''! e @y
   & @y e (x'! O y'!))}
   e {@x | (E@y : (A@z : (@z e @x == @z = @y)))}
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 11
```

```
0.  (E@y : (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 12



0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 13
```

```
0.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == z'? = y*!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 14



0.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

1.  (z'? = y*!
   -> z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 15



0.  (z'? = y*!
   -> z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))
----



0.  z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 17



0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))
----



0.  z'? = y*!

1.  z'? e {@y | (x''! e @y
```

```
   & @y e (x'! O y'!))}

2.  x''! = y'!

by lines [-1]
Next!"""

back()
back()
back()
getleft(1)

"""

Line number 14




0.  (z'? = y*!
   -> z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 15
```

```
0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

1.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  z'? = y*!

1.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 17




0.  (z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == z*? = y*!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  z'? = y*!
```

```
1.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 18




0.  (z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z*? = y*!)

1.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

3.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  z'? = y*!

1.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 19
```

```
0.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 21



0.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
```

```
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  (x''! e z*?
   & z*? e (x'! O y'!))

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 22



0.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  x''! e z*?

1.  z'? = y*!

2.  x''! = y'!
```

```
by lines [-1]
Next!"""

getleft(1)

"""

Line number 22




0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

1.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

2.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})
----



0.  x''! e z*?

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 24




0.  (z''? e {@y | (x''! e @y
```

```
   & @y e (x'! O y'!))}
   == z''? = y*!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})
----



0.  x''! e z*?

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 25



0.  (z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z''? = y*!)

1.  (z''? = y*!
   -> z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
```

```
   == @z = y*!))

3.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

4.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})
----



0.  x''! e z*?

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 26



0.  (z''? = y*!
   -> z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})
```

```
----



0.  z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  x''! e z*?

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 28



0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

1.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

2.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})
----



0.  z''? = y*!

1.  z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

2.  x''! e z*?
```

```
3.  z'? = y*!

4.  x''! = y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 28




0.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

1.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))
----



0.  z''? = y*!

1.  z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

2.  x''! e z*?

3.  z'? = y*!

4.  x''! = y'!

by lines [-1]
Next!"""
```

```
left()

"""

Line number 30



0.  (z*? = y*!
   -> z*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))
----



0.  z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  z''? = y*!

2.  z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

3.  x''! e z*?

4.  z'? = y*!

5.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 32
```

```
0.  (A@z : (@z e {@y | (x’’! e @y
   & @y e (x’! O y’!))}
   == @z = y*!))
----



0.  z*? = y*!

1.  z’? e {@y | (x’’! e @y
   & @y e (x’! O y’!))}

2.  z’’? = y*!

3.  z’’? e {@y | (x’’! e @y
   & @y e (x’! O y’!))}

4.  x’’! e z*?

5.  z’? = y*!

6.  x’’! = y’!

by lines [-1]
Next!"""

done()

"""

Line number 33




0.  y*! e {@y | (x’’! e @y
   & @y e (x’! O y’!))}

1.  (A@z : (@z e {@y | (x’’! e @y
   & @y e (x’! O y’!))}
   == @z = y*!))
----



0.  z’? e {@y | (x’’! e @y
```

```
   & @y e (x'! O y'!))}

1.  z''? = y*!

2.  z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

3.  x''! e y*!

4.  z'? = y*!

5.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 34




0.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))
----



0.  (x''! e z'?
   & z'? e (x'! O y'!))

1.  z''? = y*!

2.  z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

3.  x''! e y*!

4.  z'? = y*!
```

```
5.  x''! = y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 34




0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

1.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}
----



0.  (x''! e z'?
   & z'? e (x'! O y'!))

1.  z''? = y*!

2.  z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

3.  x''! e y*!

4.  z'? = y*!

5.  x''! = y'!

by lines [-1]
Next!"""

getleft(1)

"""
```

```
Line number 34



0.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))
----



0.  (x''! e z'?
   & z'? e (x'! O y'!))

1.  z''? = y*!

2.  z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

3.  x''! e y*!

4.  z'? = y*!

5.  x''! = y'!

by lines [-1]
Next!"""

getright(1)

"""

Line number 34



0.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
```

```
   == @z = y*!))
----



0.  z''? = y*!

1.  z''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

2.  x''! e y*!

3.  z'? = y*!

4.  x''! = y'!

5.  (x''! e z'?
   & z'? e (x'! O y'!))

by lines [-1]
Next!"""

done()

"""

Line number 31



0.  z'? = y*!

1.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))
----



0.  y*! = y*!
```

```
1.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}

2.  x''! e y*!

3.  z'? = y*!

4.  x''! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 29




0.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})
----



0.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  x''! e y*!

2.  z'? = y*!
```

```
3.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 35




0.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})
----



0.  (x''! e y*!
   & y*! e (x'! O y'!))

1.  x''! e y*!

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 36



0.  (x''! e y*!
   & y*! e (x'! O y'!))

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})
----



0.  (x''! e y*!
   & y*! e (x'! O y'!))

1.  x''! e y*!

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 37
```

```
0.  x''! e y*!

1.  y*! e (x'! O y'!)

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

3.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

4.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})
----



0.  (x''! e y*!
   & y*! e (x'! O y'!))

1.  x''! e y*!

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

getright(1)

"""

Line number 37



0.  x''! e y*!

1.  y*! e (x'! O y'!)

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
```

```
   == @z = y*!))

3.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

4.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})
----



0.  x''! e y*!

1.  z'? = y*!

2.  x''! = y'!

3.  (x''! e y*!
   & y*! e (x'! O y'!))

by lines [-1]
Next!"""

done()

"""

Line number 27



0.  y*! = y*!

1.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

3.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
```

```
   -> z'? = y*!)

4.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})
----



0.  x''! e y*!

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 27




0.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

4.  y*! = y*!
----
```

```
0.  x''! e y*!

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 38




0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

1.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

2.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

3.  y*! = y*!
----



0.  y*! = y*!

1.  x''! e y*!

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
```

```
Next!"""

done()

"""

Line number 39



0.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

4.  y*! = y*!
----



0.  x''! e y*!

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 40
```

```
0.  (x''! e y*!
   & y*! e (x'! O y'!))

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

4.  y*! = y*!
----



0.  x''! e y*!

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 41



0.  x''! e y*!

1.  y*! e (x'! O y'!)

2.  (A@z : (@z e {@y | (x''! e @y
```

```
   & @y e (x'! O y'!))}
   == @z = y*!))

3.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

4.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

5.  y*! = y*!
----



0.  x''! e y*!

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 23



0.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----
```

```
0.  y*! e (x'! O y'!)

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 42




0.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  y*! e ((x'! P x'!)
   P (x'! P y'!))

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""
```

```
right()

"""

Line number 43




0.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  y*! e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 44




0.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})
```

```
1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  (y*! = (x'! P x'!)
   V y*! = (x'! P y'!))

1.  z'? = y*!

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 45




0.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----
```

```
0.  y*! = (x'! P x'!)

1.  y*! = (x'! P y'!)

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 46




0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

1.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  y*! = y*!

1.  y*! = (x'! P x'!)

2.  y*! = (x'! P y'!)

3.  z'? = y*!

4.  x''! = y'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 48



0.  (z'*? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == z'*? = y*!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  y*! = y*!

1.  y*! = (x'! P x'!)

2.  y*! = (x'! P y'!)

3.  z'? = y*!

4.  x''! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 47



0.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}
```

```
1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  y*! = (x'! P x'!)

1.  y*! = (x'! P y'!)

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 49




0.  (x''! e y*!
   & y*! e (x'! O y'!))

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----
```

```
0.  y*! = (x'! P x'!)

1.  y*! = (x'! P y'!)

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 50




0.  x''! e y*!

1.  y*! e (x'! O y'!)

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

3.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  y*! = (x'! P x'!)

1.  y*! = (x'! P y'!)

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""
```

```
getleft(1)

"""

Line number 50



0.  y*! e (x'! O y'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  x''! e y*!
----



0.  y*! = (x'! P x'!)

1.  y*! = (x'! P y'!)

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 51



0.  y*! e ((x'! P x'!)
   P (x'! P y'!))
```

```
1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  x''! e y*!
----



0.  y*! = (x'! P x'!)

1.  y*! = (x'! P y'!)

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 52




0.  y*! e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  x''! e y*!
```

```
----



0.  y*! = (x'! P x'!)

1.  y*! = (x'! P y'!)

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 53




0.  (y*! = (x'! P x'!)
   V y*! = (x'! P y'!))

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  x''! e y*!
----



0.  y*! = (x'! P x'!)

1.  y*! = (x'! P y'!)

2.  z'? = y*!
```

```
3.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 54




0.  y*! = (x'! P x'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  x''! e y*!
----



0.  y*! = (x'! P x'!)

1.  y*! = (x'! P y'!)

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 55
```

```
0.  y*! = (x'! P y'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)

3.  x''! e y*!
----



0.  y*! = (x'! P x'!)

1.  y*! = (x'! P y'!)

2.  z'? = y*!

3.  x''! = y'!

by lines [-1]
Next!"""

getright(1)

"""

Line number 55



0.  y*! = (x'! P y'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
```

```
   -> z'? = y*!)

3.  x''! e y*!
----



0.  y*! = (x'! P y'!)

1.  z'? = y*!

2.  x''! = y'!

3.  y*! = (x'! P x'!)

by lines [-1]
Next!"""

done()

"""

Line number 20




0.  y*! = y*!

1.  (y*! = y*!
   -> y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

3.  (z'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z'? = y*!)
----



0.  z'? = y*!
```

```
1.  x''! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 16




0.  y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> y*! = y*!)
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 56




0.  (x''! e y*!
   & y*! e (x'! O y'!))

1.  (A@z : (@z e {@y | (x''! e @y
```

```
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> y*! = y*!)
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 57



0.  x''! e y*!

1.  y*! e (x'! O y'!)

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

3.  (y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> y*! = y*!)
----



0.  x''! = y'!

by lines [-1]
Next!"""

getleft(1)
```

```
"""

Line number 57



0.  y*! e (x'! O y'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> y*! = y*!)

3.  x''! e y*!
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 58



0.  y*! e ((x'! P x'!)
   P (x'! P y'!))

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> y*! = y*!)
```

```
3.  x''! e y*!
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 59



0.  y*! e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> y*! = y*!)

3.  x''! e y*!
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 60
```

```
0.  (y*! = (x'! P x'!)
   V y*! = (x'! P y'!))

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> y*! = y*!)

3.  x''! e y*!
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 61



0.  y*! = (x'! P x'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (y*! e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> y*! = y*!)

3.  x''! e y*!
----
```

```
0.  x''! = y'!

by lines [-1]
Next!"""

varelim()

"""

Line number 63




0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

1.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

2.  x''! e (x'! P x'!)
----



0.  x''! = y'!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 63
```

```
0.  x''! e (x'! P x'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 64



0.  x''! e {@x | (@x = x'! V @x = x'!)}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x''! = y'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 65




0.  (x''! = x'! V x''! = x'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 66




0.  x''! = x'!

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
```

```
2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 66




0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

1.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

2.  x''! = x'!
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 68



0.  (z*'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == z*'? = (x'! P x'!))

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 69



0.  (z*'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z*'? = (x'! P x'!))

1.  (z*'? = (x'! P x'!)
   -> z*'? e {@y | (x''! e @y
```

```
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 70




0.  (z*'? = (x'! P x'!)
   -> z*'? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
```

```
----



0.  z*'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 72




0.  (z*'? = (x'! P x'!)
   -> z*'? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----



0.  (x''! e z*'?
   & z*'? e (x'! O y'!))

1.  x''! = y'!

by lines [-1]
```

```
Next!"""

right()

"""

Line number 73




0.  (z*'? = (x'! P x'!)
   -> z*'? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----



0.  x''! e z*'?

1.  x''! = y'!

by lines [-1]
Next!"""

getleft(3)

"""

Line number 73
```

```
0.  x''! = x'!

1.  (z*'? = (x'! P x'!)
   -> z*'? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x''! e z*'?

1.  x''! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 74



0.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
```

```
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----



0.  {z** | z** = x'!}
   e (x'! O y'!)

1.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 75



0.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----
```

```
0.  {z** | z** = x'!}
   e ((x'! P x'!)
   P (x'! P y'!))

1.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 76




0.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----



0.  {z** | z** = x'!}
   e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  x''! = y'!
```

```
by lines [-1]
Next!"""

right()

"""

Line number 77




0.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----



0.  ({z** | z** = x'!}
   = (x'! P x'!)
   V {z** | z** = x'!}
   = (x'! P y'!))

1.  x''! = y'!

by lines [-1]
Next!"""

right()
```

```
"""

Line number 78



0.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----



0.  {z** | z** = x'!}
   = (x'! P x'!)

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 79
```

```
0.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----



0.  (Ax'* : (x'* e {z** | z** = x'!}
   == x'* e (x'! P x'!)))

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 80



0.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
```

```
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----



0.  (x*'! e {z** | z** = x'!}
   == x*'! e (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 81



0.  x*'! e {z** | z** = x'!}

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})
```

```
2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x*'! e (x'! P x'!)

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 83




0.  x*'! = x'!

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
```

```
3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x*'! e (x'! P x'!)

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 84



0.  x*'! = x'!

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
```

```
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x*'! e {@x | (@x = x'! V @x = x'!)}

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 85



0.  x*'! = x'!

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----
```

```
0.  (x*'! = x'! V x*'! = x'!)

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 86




0.  x*'! = x'!

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x*'! = x'!
```

```
1.  x*'! = x'!

2.  {z** | z** = x'!}
   = (x'! P y'!)

3.  x''! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 82




0.  x*'! e (x'! P x'!)

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x*'! e {z** | z** = x'!}

1.  {z** | z** = x'!}
```

```
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 87




0.  x*'! e (x'! P x'!)

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x*'! = x'!

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 88



0.  x*'! e {@x | (@x = x'! V @x = x'!)}

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x*'! = x'!

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 89



0.  (x*'! = x'! V x*'! = x'!)

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x*'! = x'!

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 90
```

```
0.  x*'! = x'!

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x*'! = x'!

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 91
```

```
0.  x*'! = x'!

1.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x*'! = x'!

1.  {z** | z** = x'!}
   = (x'! P y'!)

2.  x''! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 71



0.  {z** | z** = x'!}
   = (x'! P x'!)

1.  ({z** | z** = x'!}
```

```
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x''! = y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 71



0.  ({z** | z** = x'!}
   = (x'! P x'!)
   -> {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
```

```
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!

4.  {z** | z** = x'!}
   = (x'! P x'!)
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 92



0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

1.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

2.  x''! = x'!

3.  {z** | z** = x'!}
   = (x'! P x'!)
----



0.  {z** | z** = x'!}
   = (x'! P x'!)
```

```
1.  x''! = y'!

by lines [-1]
Next!"""

getleft(3)

"""

Line number 92




0.  {z** | z** = x'!}
   = (x'! P x'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
----



0.  {z** | z** = x'!}
   = (x'! P x'!)

1.  x''! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 93
```

```
0.  {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!

4.  {z** | z** = x'!}
   = (x'! P x'!)
----



0.  x''! = y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 93



0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

1.  ((x'! P x'!)
   e {@y | (x''! e @y
```

```
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

2.  x''! = x'!

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 94



0.  (z'''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == z'''? = (x'! P x'!))

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
```

```
4.  {z** | z** = x'!}
   = (x'! P x'!)

5.  {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
----



0.  x''! = y'!

by lines [-1]
Next!"""

getleft(3)

"""

Line number 94



0.  x''! = x'!

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}

3.  (z'''? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == z'''? = (x'! P x'!))

4.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
```

```
----



0.  x''! = y'!

by lines [-1]
Next!"""

varelim()

"""

Line number 95



0.  {z** | z** = x'!}
   = (x'! P x'!)

1.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

2.  (z'''? e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == z'''? = (x'! P x'!))

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""
```

```
getleft(2)

"""

Line number 95



0.  (z’’’? e {@y | (x’! e @y
   & @y e (x’! O y’!))}
   == z’’’? = (x’! P x’!))

1.  (A@z : (@z e {@y | (x’! e @y
   & @y e (x’! O y’!))}
   == @z = (x’! P x’!)))

2.  ((x’! P x’!)
   e {@y | (x’! e @y
   & @y e (x’! O y’!))}
   -> (x’! P x’!)
   = (x’! P x’!))

3.  {z** | z** = x’!}
   = (x’! P x’!)

4.  {z** | z** = x’!}
   e {@y | (x’! e @y
   & @y e (x’! O y’!))}
----



0.  x’! = y’!

by lines [-1]
Next!"""

left()

"""

Line number 96
```

```
0.  (z’’’? e {@y | (x’! e @y
   & @y e (x’! O y’!))}
   -> z’’’? = (x’! P x’!))

1.  (z’’’? = (x’! P x’!)
   -> z’’’? e {@y | (x’! e @y
   & @y e (x’! O y’!))})

2.  (A@z : (@z e {@y | (x’! e @y
   & @y e (x’! O y’!))}
   == @z = (x’! P x’!)))

3.  ((x’! P x’!)
   e {@y | (x’! e @y
   & @y e (x’! O y’!))}
   -> (x’! P x’!)
   = (x’! P x’!))

4.  {z** | z** = x’!}
   = (x’! P x’!)

5.  {z** | z** = x’!}
   e {@y | (x’! e @y
   & @y e (x’! O y’!))}
----



0.  x’! = y’!

by lines [-1]
Next!"""

left()

"""

Line number 97



0.  (z’’’? = (x’! P x’!)
   -> z’’’? e {@y | (x’! e @y
```

```
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  z'''? e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 99



0.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

1.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
```

```
   -> (x'! P x'!)
   = (x'! P x'!))

2.  {z** | z** = x'!}
   = (x'! P x'!)

3.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  z'''? = (x'! P x'!)

1.  z'''? e {@y | (x'! e @y
   & @y e (x'! O y'!))}

2.  x'! = y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 99




0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
```

```
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  z'''? = (x'! P x'!)

1.  z'''? e {@y | (x'! e @y
   & @y e (x'! O y'!))}

2.  x'! = y'!

by lines [-1]
Next!"""

getright(1)

"""

Line number 99




0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  z'''? e {@y | (x'! e @y
```

```
   & @y e (x'! O y'!))}

1.  x'! = y'!

2.  z'''? = (x'! P x'!)

by lines [-1]
Next!"""

right()

"""

Line number 101




0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  (x'! e z'''?
   & z'''? e (x'! O y'!))

1.  x'! = y'!

2.  z'''? = (x'! P x'!)

by lines [-1]
```

```
Next!"""

right()

"""

Line number 102



0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  x'! e z'''?

1.  x'! = y'!

2.  z'''? = (x'! P x'!)

by lines [-1]
Next!"""

skip()

"""

Line number 103
```

```
0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  z'''? e (x'! O y'!)

1.  x'! = y'!

2.  z'''? = (x'! P x'!)

by lines [-1]
Next!"""

right()

"""

Line number 104



0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
```

```
1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  z'''? e ((x'! P x'!)
   P (x'! P y'!))

1.  x'! = y'!

2.  z'''? = (x'! P x'!)

by lines [-1]
Next!"""

right()

"""

Line number 105



0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
```

```
3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  z'''? e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  x'! = y'!

2.  z'''? = (x'! P x'!)

by lines [-1]
Next!"""

right()

"""

Line number 106



0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----
```

```
0.  (z'''? = (x'! P x'!)
   V z'''? = (x'! P y'!))

1.  x'! = y'!

2.  z'''? = (x'! P x'!)

by lines [-1]
Next!"""

right()

"""

Line number 107




0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  z'''? = (x'! P x'!)

1.  z'''? = (x'! P y'!)

2.  x'! = y'!
```

```
3.  z’’’? = (x’! P x’!)

by lines [-1]
Next!"""

getright(1)

"""

Line number 107




0.  ((x’! P x’!)
   e {@y | (x’! e @y
   & @y e (x’! O y’!))}
   -> (x’! P x’!)
   = (x’! P x’!))

1.  {z** | z** = x’!}
   = (x’! P x’!)

2.  {z** | z** = x’!}
   e {@y | (x’! e @y
   & @y e (x’! O y’!))}

3.  (A@z : (@z e {@y | (x’! e @y
   & @y e (x’! O y’!))}
   == @z = (x’! P x’!)))
----



0.  z’’’? = (x’! P y’!)

1.  x’! = y’!

2.  z’’’? = (x’! P x’!)

3.  z’’’? = (x’! P x’!)

by lines [-1]
Next!"""

done()
```

```
"""

Line number 100



0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 108
```

```
0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  (x'! e (x'! P y'!)
   & (x'! P y'!)
   e (x'! O y'!))

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 109
```

```
0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! e (x'! P y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 111



0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
```

```
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! e {@x | (@x = x'! V @x = y'!)}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 112



0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
```

```
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  (x'! = x'! V x'! = y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 113




0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
```

```
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = x'!

1.  x'! = y'!

2.  x'! = y'!

by lines [-1]
Next!"""

done()

"""

Line number 110



0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----
```

```
0.  (x'! P y'!)
   e (x'! O y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 114




0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  (x'! P y'!)
   e ((x'! P x'!)
   P (x'! P y'!))
```

```
1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 115




0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  (x'! P y'!)
   e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  x'! = y'!

by lines [-1]
```

```
Next!"""

right()

"""

Line number 116



0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  ((x'! P y'!)
   = (x'! P x'!)
   V (x'! P y'!)
   = (x'! P y'!))

1.  x'! = y'!

by lines [-1]
Next!"""

right()
```

```
"""

Line number 117



0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  (x'! P y'!)
   = (x'! P x'!)

1.  (x'! P y'!)
   = (x'! P y'!)

2.  x'! = y'!

by lines [-1]
Next!"""

getright(1)

"""
```

```
Line number 117



0.  (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  {z** | z** = x'!}
   = (x'! P x'!)

4.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  (x'! P y'!)
   = (x'! P y'!)

1.  x'! = y'!

2.  (x'! P y'!)
   = (x'! P x'!)

by lines [-1]
Next!"""

done()

"""

Line number 98
```

```
0.  (x'! P y'!)
   = (x'! P x'!)

1.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  {z** | z** = x'!}
   = (x'! P x'!)

5.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 118
```

```
0.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

1.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  {z** | z** = x'!}
   = (x'! P x'!)

5.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 119



0.  ((x'! P y'!) e x'''?
   == (x'! P x'!) e x'''?)
```

```
1.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

5.  {z** | z** = x'!}
   = (x'! P x'!)

6.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = y'!

by lines [-1]
Next!"""

setunknown ("x'''","{ue'yu")

"""

Line number 119



0.  ((x'! P y'!)
   e {u | y'! e u}
```

```
   == (x'! P x'!)
   e {u | y'! e u})

1.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

5.  {z** | z** = x'!}
   = (x'! P x'!)

6.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 120
```

```
0.  ((x'! P y'!)
   e {u | y'! e u}
   -> (x'! P x'!)
   e {u | y'! e u})

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

6.  {z** | z** = x'!}
   = (x'! P x'!)

7.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 121



0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

5.  {z** | z** = x'!}
   = (x'! P x'!)

6.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  (x'! P y'!)
   e {u | y'! e u}

1.  x'! = y'!
```

```
by lines [-1]
Next!"""

right()

"""

Line number 123




0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

5.  {z** | z** = x'!}
   = (x'! P x'!)

6.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----
```

```
0.  y'! e (x'! P y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 124



0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

5.  {z** | z** = x'!}
   = (x'! P x'!)

6.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
```

```
----



0.  y'! e {@x | (@x = x'! V @x = y'!)}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 125




0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

5.  {z** | z** = x'!}
   = (x'! P x'!)
```

```
6.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  (y'! = x'! V y'! = y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 126



0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
```

```
   = (x'! P x'!))

5.  {z** | z** = x'!}
   = (x'! P x'!)

6.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  y'! = x'!

1.  y'! = y'!

2.  x'! = y'!

by lines [-1]
Next!"""

getright(1)

"""

Line number 126



0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
```

```
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

5.  {z** | z** = x'!}
   = (x'! P x'!)

6.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  y'! = y'!

1.  x'! = y'!

2.  y'! = x'!

by lines [-1]
Next!"""

done()

"""

Line number 122




0.  (x'! P x'!)
   e {u | y'! e u}

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))
```

```
3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

6.  {z** | z** = x'!}
   = (x'! P x'!)

7.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 127



0.  y'! e (x'! P x'!)

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
```

```
   e {u | y'! e u})

2.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

6.  {z** | z** = x'!}
   = (x'! P x'!)

7.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 128



0.  y'! e {@x | (@x = x'! V @x = x'!)}
```

```
1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

6.  {z** | z** = x'!}
   = (x'! P x'!)

7.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 129
```

```
0.  (y'! = x'! V y'! = x'!)

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

6.  {z** | z** = x'!}
   = (x'! P x'!)

7.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 130



0.  y'! = x'!

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

6.  {z** | z** = x'!}
   = (x'! P x'!)

7.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----



0.  x'! = y'!

by lines [-1]
```

```
Next!"""

varelim()

"""

Line number 132



0.  ((x'! P x'!)
   e {u | x'! e u}
   -> (x'! P x'!)
   e {u | x'! e u})

1.  (Ax** : ((x'! P x'!) e x**
   == (x'! P x'!) e x**))

2.  ((x'! P x'!)
   = (x'! P x'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

5.  {z** | z** = x'!}
   = (x'! P x'!)

6.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O x'!))}
----



0.  x'! = x'!
```

```
by lines [-1]
Next!"""

done()

"""

Line number 131




0.  y'! = x'!

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax** : ((x'! P y'!) e x**
   == (x'! P x'!) e x**))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

6.  {z** | z** = x'!}
   = (x'! P x'!)

7.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
----
```

```
0.  x'! = y'!

by lines [-1]
Next!"""

varelim()

"""

Line number 133




0.  ((x'! P x'!)
   e {u | x'! e u}
   -> (x'! P x'!)
   e {u | x'! e u})

1.  (Ax** : ((x'! P x'!) e x**
   == (x'! P x'!) e x**))

2.  ((x'! P x'!)
   = (x'! P x'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

5.  {z** | z** = x'!}
   = (x'! P x'!)

6.  {z** | z** = x'!}
   e {@y | (x'! e @y
```

```
   & @y e (x'! O x'!))}
----



0.  x'! = x'!

by lines [-1]
Next!"""

done()

"""

Line number 67



0.  x''! = x'!

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 67
```

```
0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

1.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

2.  x''! = x'!
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 134




0.  (z''*? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == z''*? = (x'! P x'!))

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

3.  x''! = x'!
```

```
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 135



0.  (z''*? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z''*? = (x'! P x'!))

1.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  x''! = x'!
----



0.  x''! = y'!

by lines [-1]
Next!"""
```

```
getleft(4)

"""

Line number 135



0.  x''! = x'!

1.  (z''*? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> z''*? = (x'! P x'!))

2.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x''! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

varelim()

"""

Line number 136
```

```
0.  (z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> z''*? = (x'! P x'!))

1.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 137



0.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
```

```
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 139



0.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  (x'! e z''*?
   & z''*? e (x'! O y'!))

1.  x'! = y'!

by lines [-1]
```

```
Next!"""

right()

"""

Line number 140



0.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! e z''*?

1.  x'! = y'!

by lines [-1]
Next!"""

skip()

"""

Line number 141



0.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x'! e @y
```

```
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  z''*? e (x'! O y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 142



0.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----
```

```
0.  z''*? e ((x'! P x'!)
   P (x'! P y'!))

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 143




0.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  z''*? e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  x'! = y'!

by lines [-1]
Next!"""

right()
```

```
"""

Line number 144



0.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  (z''*? = (x'! P x'!)
   V z''*? = (x'! P y'!))

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 145



0.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
```

```
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  z''*? = (x'! P x'!)

1.  z''*? = (x'! P y'!)

2.  x'! = y'!

by lines [-1]
Next!"""

getright(1)

"""

Line number 145



0.  (z''*? = (x'! P x'!)
   -> z''*? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----
```

```
0.  z''*? = (x'! P y'!)

1.  x'! = y'!

2.  z''*? = (x'! P x'!)

by lines [-1]
Next!"""

done()

"""

Line number 138




0.  (x'! P y'!)
   = (x'! P x'!)

1.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""
```

```
left()

"""

Line number 146



0.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

1.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 147



0.  ((x'! P y'!) e x'*'?
```

```
   == (x'! P x'!) e x'*'?)

1.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

setunknown ("x'*'","{ue'yu")

"""

Line number 147



0.  ((x'! P y'!)
   e {u | y'! e u}
   == (x'! P x'!)
   e {u | y'! e u})

1.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))
```

```
2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 148



0.  ((x'! P y'!)
   e {u | y'! e u}
   -> (x'! P x'!)
   e {u | y'! e u})

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))
```

```
3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 149



0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})
```

```
3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  (x'! P y'!)
   e {u | y'! e u}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 151



0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
```

```
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  y'! e (x'! P y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 152



0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
```

```
4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  y'! e {@x | (@x = x'! V @x = y'!)}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 153



0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
```

```
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  (y'! = x'! V y'! = y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 154



0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----
```

```
0.  y'! = x'!

1.  y'! = y'!

2.  x'! = y'!

by lines [-1]
Next!"""

getright(1)

"""

Line number 154




0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

2.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----
```

```
0.  y'! = y'!

1.  x'! = y'!

2.  y'! = x'!

by lines [-1]
Next!"""

done()

"""

Line number 150




0.  (x'! P x'!)
   e {u | y'! e u}

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
```

```
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 155



0.  y'! e (x'! P x'!)

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----
```

```
0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 156



0.  y'! e {@x | (@x = x'! V @x = x'!)}

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 157



0.  (y'! = x'! V y'! = x'!)

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""
```

```
left()

"""

Line number 158



0.  y'! = x'!

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

varelim()

"""
```

```
Line number 160



0.  ((x'! P x'!)
   e {u | x'! e u}
   -> (x'! P x'!)
   e {u | x'! e u})

1.  (Ax''* : ((x'! P x'!) e x''*
   == (x'! P x'!) e x''*))

2.  ((x'! P x'!)
   = (x'! P x'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = x'!

by lines [-1]
Next!"""

done()

"""

Line number 159
```

```
0.  y'! = x'!

1.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

2.  (Ax''* : ((x'! P y'!) e x''*
   == (x'! P x'!) e x''*))

3.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

5.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

varelim()

"""

Line number 161



0.  ((x'! P x'!)
   e {u | x'! e u}
```

```
   -> (x'! P x'!)
   e {u | x'! e u})

1.  (Ax''* : ((x'! P x'!) e x''*
   == (x'! P x'!) e x''*))

2.  ((x'! P x'!)
   = (x'! P x'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   == @z = (x'! P x'!)))

4.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = x'!

by lines [-1]
Next!"""

done()

"""

Line number 62



0.  y*! = (x'! P y'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = y*!))

2.  (y*! e {@y | (x''! e @y
```

```
   & @y e (x'! O y'!))}
   -> y*! = y*!)

3.  x''! e y*!
----



0.  x''! = y'!

by lines [-1]
Next!"""

varelim()

"""

Line number 162



0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

1.  ((x'! P y'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

2.  x''! e (x'! P y'!)
----



0.  x''! = y'!

by lines [-1]
Next!"""

getleft(2)

"""
```

```
Line number 162



0.  x''! e (x'! P y'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 163



0.  x''! e {@x | (@x = x'! V @x = y'!)}

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----
```

```
0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 164




0.  (x''! = x'! V x''! = y'!)

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 165
```

```
0.  x''! = x'!

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

skip()

"""

Line number 166




0.  x''! = y'!

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x''! = y'!
```

```
by lines [-1]
Next!"""

done()

"""

Line number 6



0.  x''! e {u | u = y'!}
----



0.  x''! e P*(a:(x'! O y'!))

by lines [-1]
Next!"""

skip()

"""

Line number 102



0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
```

```
   == @z = (x'! P x'!)))
----



0.  x'! e (x'! P y'!)

1.  x'! = y'!

2.  (x'! P y'!)
   = (x'! P x'!)

by lines [-1]
Next!"""

right()

"""

Line number 167



0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  x'! e {@x | (@x = x'! V @x = y'!)}
```

```
1.  x'! = y'!

2.  (x'! P y'!)
   = (x'! P x'!)

by lines [-1]
Next!"""

right()

"""

Line number 168




0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  (x'! = x'! V x'! = y'!)

1.  x'! = y'!

2.  (x'! P y'!)
   = (x'! P x'!)

by lines [-1]
Next!"""
```

```
right()

"""

Line number 169



0.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

1.  {z** | z** = x'!}
   = (x'! P x'!)

2.  {z** | z** = x'!}
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))
----



0.  x'! = x'!

1.  x'! = y'!

2.  x'! = y'!

3.  (x'! P y'!)
   = (x'! P x'!)

by lines [-1]
Next!"""

done()

"""
```

```
Line number 140



0.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! e (x'! P y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 170



0.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})
```

```
1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! e {@x | (@x = x'! V @x = y'!)}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 171



0.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----
```

```
0.  (x'! = x'! V x'! = y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 172




0.  ((x'! P y'!)
   = (x'! P x'!)
   -> (x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P x'!)))

2.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P x'!)
   = (x'! P x'!))
----



0.  x'! = x'!

1.  x'! = y'!

2.  x'! = y'!

by lines [-1]
Next!"""
```

```
done()

"""

Line number 165




0.  x''! = x'!

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 165




0.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

1.  ((x'! P y'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
```

```
   = (x'! P y'!))

2.  x''! = x'!
----



0.  x''! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 173



0.  (z'*'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == z'*'? = (x'! P y'!))

1.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

3.  x''! = x'!
----



0.  x''! = y'!

by lines [-1]
Next!"""

getleft(3)
```

```
"""

Line number 173



0.  x''! = x'!

1.  (z'*'? e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == z'*'? = (x'! P y'!))

2.  (A@z : (@z e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

3.  ((x'! P y'!)
   e {@y | (x''! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x''! = y'!

by lines [-1]
Next!"""

varelim()

"""

Line number 174



0.  (z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == z'*'? = (x'! P y'!))

1.  (A@z : (@z e {@y | (x'! e @y
```

```
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 175



0.  (z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> z'*'? = (x'! P y'!))

1.  (z'*'? = (x'! P y'!)
   -> z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

3.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----
```

```
0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 176




0.  (z'*'? = (x'! P y'!)
   -> z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 178
```

```
0.  (z'*'? = (x'! P y'!)
   -> z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  (x'! e z'*'?
   & z'*'? e (x'! O y'!))

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 179



0.  (z'*'? = (x'! P y'!)
   -> z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
```

```
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x'! e z'*'?

1.  x'! = y'!

by lines [-1]
Next!"""

skip()

"""

Line number 180



0.  (z'*'? = (x'! P y'!)
   -> z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  z'*'? e (x'! O y'!)

1.  x'! = y'!

by lines [-1]
```

```
Next!"""

right()

"""

Line number 181




0.  (z'*'? = (x'! P y'!)
   -> z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  z'*'? e ((x'! P x'!)
   P (x'! P y'!))

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 182




0.  (z'*'? = (x'! P y'!)
```

```
   -> z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  z'*'? e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 183



0.  (z'*'? = (x'! P y'!)
   -> z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
```

```
----



0.  (z'*'? = (x'! P x'!)
   V z'*'? = (x'! P y'!))

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 184




0.  (z'*'? = (x'! P y'!)
   -> z'*'? e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  z'*'? = (x'! P x'!)

1.  z'*'? = (x'! P y'!)

2.  x'! = y'!

by lines [-1]
Next!"""
```

```
done()

"""

Line number 177



0.  (x'! P x'!)
   = (x'! P y'!)

1.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

3.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 185
```

```
0.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

1.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

3.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 186



0.  ((x'! P x'!) e x*''?
   == (x'! P y'!) e x*''?)

1.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

2.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
```

```
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

4.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 187



0.  ((x'! P x'!) e x*''?
   -> (x'! P y'!) e x*''?)

1.  ((x'! P y'!) e x*''?
   -> (x'! P x'!) e x*''?)

2.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

3.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
```

```
   == @z = (x'! P y'!)))

5.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

back()
setunknown ("x*''","{ue'yu")

"""

Line number 187



0.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})

1.  ((x'! P y'!)
   e {u | y'! e u}
   -> (x'! P x'!)
   e {u | y'! e u})

2.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

3.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

4.  (A@z : (@z e {@y | (x'! e @y
```

```
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

5.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x'! = y'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 187



0.  ((x'! P y'!)
   e {u | y'! e u}
   -> (x'! P x'!)
   e {u | y'! e u})

1.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

2.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

4.  ((x'! P y'!)
   e {@y | (x'! e @y
```

```
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

5.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 188



0.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

1.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

3.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

4.  ((x'! P x'!)
```

```
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  (x'! P y'!)
   e {u | y'! e u}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 190



0.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

1.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

3.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

4.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
```

```
   e {u | y'! e u})
----



0.  y'! e (x'! P y'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 191



0.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

1.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

3.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

4.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----
```

```
0.  y'! e {@x | (@x = x'! V @x = y'!)}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 192



0.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

1.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

3.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

4.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  (y'! = x'! V y'! = y'!)
```

```
1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 193




0.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

1.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

3.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

4.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  y'! = x'!

1.  y'! = y'!
```

```
2.  x'! = y'!

by lines [-1]
Next!"""

getright(1)

"""

Line number 193



0.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

1.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

3.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

4.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  y'! = y'!

1.  x'! = y'!

2.  y'! = x'!
```

```
by lines [-1]
Next!"""

done()

"""

Line number 189




0.  (x'! P x'!)
   e {u | y'! e u}

1.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

2.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

4.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

5.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  x'! = y'!

by lines [-1]
```

```
Next!"""

left()

"""

Line number 194



0.  y'! e (x'! P x'!)

1.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

2.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

4.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

5.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 195



0.  y'! e {@x | (@x = x'! V @x = x'!)}

1.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

2.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

4.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

5.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""
```

```
Line number 196



0.  (y'! = x'! V y'! = x'!)

1.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

2.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

4.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

5.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  x'! = y'!

by lines [-1]
Next!"""

left()

"""

Line number 197
```

```
0.  y'! = x'!

1.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

2.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

4.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

5.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  x'! = y'!

by lines [-1]
Next!"""

varelim()

"""

Line number 199



0.  (Ax'** : ((x'! P x'!) e x'**
```

```
   == (x'! P x'!) e x'**))

1.  ((x'! P x'!)
   = (x'! P x'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))})

2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  ((x'! P x'!)
   e {u | x'! e u}
   -> (x'! P x'!)
   e {u | x'! e u})
----



0.  x'! = x'!

by lines [-1]
Next!"""

done()

"""

Line number 198



0.  y'! = x'!

1.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P y'!) e x'**))

2.  ((x'! P x'!)
```

```
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

3.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

4.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))

5.  ((x'! P x'!)
   e {u | y'! e u}
   -> (x'! P y'!)
   e {u | y'! e u})
----



0.  x'! = y'!

by lines [-1]
Next!"""

varelim()

"""

Line number 200



0.  (Ax'** : ((x'! P x'!) e x'**
   == (x'! P x'!) e x'**))

1.  ((x'! P x'!)
   = (x'! P x'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))})
```

```
2.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   == @z = (x'! P x'!)))

3.  ((x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   -> (x'! P x'!)
   = (x'! P x'!))

4.  ((x'! P x'!)
   e {u | x'! e u}
   -> (x'! P x'!)
   e {u | x'! e u})
----



0.  x'! = x'!

by lines [-1]
Next!"""

done()

"""

Line number 6



0.  x''! e {u | u = y'!}
----



0.  x''! e P*(a:(x'! O y'!))

by lines [-1]
Next!"""

skip()

"""
```

```
Line number 179



0.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x'! e (x'! P x'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 201



0.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})
```

```
1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x'! e {@x | (@x = x'! V @x = x'!)}

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 202



0.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----
```

```
0.  (x'! = x'! V x'! = x'!)

1.  x'! = y'!

by lines [-1]
Next!"""

right()

"""

Line number 203




0.  ((x'! P x'!)
   = (x'! P y'!)
   -> (x'! P x'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))})

1.  (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   == @z = (x'! P y'!)))

2.  ((x'! P y'!)
   e {@y | (x'! e @y
   & @y e (x'! O y'!))}
   -> (x'! P y'!)
   = (x'! P y'!))
----



0.  x'! = x'!

1.  x'! = x'!

2.  x'! = y'!

by lines [-1]
Next!"""
```

```
done()

"""

Line number 6



0.  x''! e {u | u = y'!}
----



0.  x''! e P*(a:(x'! O y'!))

by lines [-1]
Next!"""


"""

Line number 6



0.  x''! e {u | u = y'!}
----



0.  x''! e P*(a:(x'! O y'!))

by lines [-1]
Next!"""

left()

"""

Line number 204
```

```
0.  x''! = y'!
----



0.  x''! e P*(a:(x'! O y'!))

by lines [-1]
Next!"""

varelim()

"""

Line number 205



----



0.  y'! e P*(a:(x'! O y'!))

by lines [-1]
Next!"""

right()

"""

Line number 206



----



0.  y'! e {@x | {@y | (@x e @y
   & @y e (x'! O y'!))} e U}

by lines [-1]
Next!"""

right()
```

```
"""

Line number 207



----



0.  {@y | (y'! e @y
   & @y e (x'! O y'!))} e U

by lines [-1]
Next!"""

right()

"""

Line number 208



----



0.  {@y | (y'! e @y
   & @y e (x'! O y'!))}
   e {@x | (E@y : (A@z : (@z e @x == @z = @y)))}

by lines [-1]
Next!"""

right()

"""

Line number 209



----
```

```
0.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

right()

"""

Line number 210



----



0.  (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = y''?))

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

right()

"""

Line number 211



----



0.  (z'**! e {@y | (y'! e @y
   & @y e (x'! O y'!))}
```

```
   == z'**! = y''?)

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

right()

"""

Line number 212




0.  z'**! e {@y | (y'! e @y
   & @y e (x'! O y'!))}
----



0.  z'**! = y''?

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

left()

"""

Line number 214




0.  (y'! e z'**!
   & z'**! e (x'! O y'!))
----
```

```
0.  z'**! = y''?

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

left()

"""

Line number 215




0.  y'! e z'**!

1.  z'**! e (x'! O y'!)
----



0.  z'**! = y''?

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

getleft(1)

"""

Line number 215




0.  z'**! e (x'! O y'!)
```

```
1.  y'! e z'**!
----



0.  z'**! = y''?

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

left()

"""

Line number 216




0.  z'**! e ((x'! P x'!)
   P (x'! P y'!))

1.  y'! e z'**!
----



0.  z'**! = y''?

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

left()

"""

Line number 217
```

```
0.  z'**! e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  y'! e z'**!
----



0.  z'**! = y''?

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

left()

"""

Line number 218




0.  (z'**! = (x'! P x'!)
   V z'**! = (x'! P y'!))

1.  y'! e z'**!
----



0.  z'**! = y''?

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""
```

```
left()

"""

Line number 219



0.  z'**! = (x'! P x'!)

1.  y'! e z'**!
----



0.  z'**! = y''?

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

skip()

"""

Line number 220



0.  z'**! = (x'! P y'!)

1.  y'! e z'**!
----



0.  z'**! = y''?

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
```

```
   == @z = @y)))

by lines [-1]
Next!"""

done()

"""

Line number 213




0.  z'**! = (x'! P y'!)
----



0.  z'**! e {@y | (y'! e @y
   & @y e (x'! O y'!))}

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

varelim()

"""

Line number 221



----



0.  (x'! P y'!)
   e {@y | (y'! e @y
   & @y e (x'! O y'!))}

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
```

```
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

right()

"""

Line number 222



----



0.  (y'! e (x'! P y'!)
   & (x'! P y'!)
   e (x'! O y'!))

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

right()

"""

Line number 223



----



0.  y'! e (x'! P y'!)

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))
```

```
by lines [-1]
Next!"""

right()

"""

Line number 225



----



0.  y'! e {@x | (@x = x'! V @x = y'!)}

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

right()

"""

Line number 226



----



0.  (y'! = x'! V y'! = y'!)

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""
```

```
right()

"""

Line number 227



----



0.  y'! = x'!

1.  y'! = y'!

2.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

getright(1)

"""

Line number 227



----



0.  y'! = y'!

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

2.  y'! = x'!

by lines [-1]
Next!"""
```

```
done()

"""

Line number 224



----



0.  (x'! P y'!)
   e (x'! O y'!)

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

right()

"""

Line number 228



----



0.  (x'! P y'!)
   e ((x'! P x'!)
   P (x'! P y'!))

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

right()
```

```
"""

Line number 229



----



0.  (x'! P y'!)
   e {@x | (@x = (x'! P x'!)
   V @x = (x'! P y'!))}

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

right()

"""

Line number 230



----



0.  ((x'! P y'!)
   = (x'! P x'!)
   V (x'! P y'!)
   = (x'! P y'!))

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""
```

```
right()

"""

Line number 231



----



0.  (x'! P y'!)
   = (x'! P x'!)

1.  (x'! P y'!)
   = (x'! P y'!)

2.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

getright(1)

"""

Line number 231



----



0.  (x'! P y'!)
   = (x'! P y'!)

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

2.  (x'! P y'!)
   = (x'! P x'!)
```

```
by lines [-1]
Next!"""

done()

"""

Line number 219




0.  z'**! = (x'! P x'!)

1.  y'! e z'**!
----



0.  z'**! = (x'! P y'!)

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

varelim()

"""

Line number 232




0.  y'! e (x'! P x'!)
----



0.  (x'! P x'!)
   = (x'! P y'!)
```

```
1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

left()

"""

Line number 233



0.  y'! e {@x | (@x = x'! V @x = x'!)}
----



0.  (x'! P x'!)
   = (x'! P y'!)

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

left()

"""

Line number 234



0.  (y'! = x'! V y'! = x'!)
----



0.  (x'! P x'!)
```

```
   = (x'! P y'!)

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

left()

"""

Line number 235




0.  y'! = x'!
----



0.  (x'! P x'!)
   = (x'! P y'!)

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

varelim()

"""

Line number 237



----



0.  (x'! P x'!)
```

```
   = (x'! P x'!)

1.  (E@y : (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   == @z = @y)))

by lines [-1]
Next!"""

done()

"""

Line number 236




0.  y'! = x'!
----



0.  (x'! P x'!)
   = (x'! P y'!)

1.  (E@y : (A@z : (@z e {@y | (y'! e @y
   & @y e (x'! O y'!))}
   == @z = @y)))

by lines [-1]
Next!"""

varelim()

"""

Line number 238



----



0.  (x'! P x'!)
```

```
   = (x'! P x'!)

1.  (E@y : (A@z : (@z e {@y | (x'! e @y
   & @y e (x'! O x'!))}
   == @z = @y)))

by lines [-1]
Next!"""

done()

"""Done!"""
savetheorem("Proj2")
```

3.7 Foundations of arithmetic

Also a large file. Some work on definitions and axioms for arithmetic.

```
from graph2 import *
#setlog("editarith")

deft ("Z", "{xAy~eyx")
deft ("N", "{x~=xx")
start ("AxXexZ=xN")

"""

Line number 0



----



0.  (Ax : (x e Z == x = N))

by lines [-1]
Next!"""

right()

"""

Line number 1
```

```
----



0.  (x'! e Z == x'! = N)

by lines [-1]
Next!"""

right()

"""

Line number 2




0.  x'! e Z
----



0.  x'! = N

by lines [-1]
Next!"""

right()

"""

Line number 4




0.  x'! e Z
----



0.  (Ax* : (x* e x'! == x* e N))
```

```
by lines [-1]
Next!"""

right()

"""

Line number 5



0.  x'! e Z
----



0.  (x''! e x'! == x''! e N)

by lines [-1]
Next!"""

right()

"""

Line number 6



0.  x''! e x'!

1.  x'! e Z
----



0.  x''! e N

by lines [-1]
Next!"""

right()

"""
```

```
Line number 8



0.  x''! e x'!

1.  x'! e Z
----



0.  x''! e {@x | ~@x = @x}

by lines [-1]
Next!"""

right()

"""

Line number 9



0.  x''! e x'!

1.  x'! e Z
----



0.  ~x''! = x''!

by lines [-1]
Next!"""

right()

"""

Line number 10
```

```
0.  x''! = x''!

1.  x''! e x'!

2.  x'! e Z
----



by lines [-1]
Next!"""

getleft(2)

"""

Line number 10



0.  x'! e Z

1.  x''! = x''!

2.  x''! e x'!
----



by lines [-1]
Next!"""

left()

"""

Line number 11



0.  x'! e {@x | (A@y : ~@y e @x)}
```

```
1.  x''! = x''!

2.  x''! e x'!
----



by lines [-1]
Next!"""

left()

"""

Line number 12



0.  (A@y : ~@y e x'!)

1.  x''! = x''!

2.  x''! e x'!
----



by lines [-1]
Next!"""

left()

"""

Line number 13



0.  ~y'? e x'!

1.  (A@y : ~@y e x'!)

2.  x''! = x''!
```

```
3.  x''! e x'!
----



by lines [-1]
Next!"""

left()

"""

Line number 14




0.  (A@y : ~@y e x'!)

1.  x''! = x''!

2.  x''! e x'!
----



0.  y'? e x'!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 14




0.  x''! e x'!

1.  (A@y : ~@y e x'!)

2.  x''! = x''!
----
```

```
0.  y'? e x'!

by lines [-1]
Next!"""

done()

"""

Line number 7



0.  x''! e N

1.  x'! e Z
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 15



0.  x''! e {@x | ~@x = @x}

1.  x'! e Z
----



0.  x''! e x'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 16



0.  ~x''! = x''!

1.  x'! e Z
----



0.  x''! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 17



0.  x'! e Z
----



0.  x''! = x''!

1.  x''! e x'!

by lines [-1]
Next!"""
```

```
done()

"""

Line number 3



0.  x'! = N
----



0.  x'! e Z

by lines [-1]
Next!"""

right()

"""

Line number 18



0.  x'! = N
----



0.  x'! e {@x | (A@y : ~@y e @x)}

by lines [-1]
Next!"""

right()

"""

Line number 19
```

```
0.  x'! = N
----



0.  (A@y : ~@y e x'!)

by lines [-1]
Next!"""

right()

"""

Line number 20



0.  x'! = N
----



0.  ~y*! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 21



0.  (Ax'* : (x'! e x'* == N e x'*))
----



0.  ~y*! e x'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 22



0.  (x'! e x*'? == N e x*'?)

1.  (Ax'* : (x'! e x'* == N e x'*))
----



0.  ~y*! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 23



0.  (x'! e x*'? -> N e x*'?)

1.  (N e x*'? -> x'! e x*'?)

2.  (Ax'* : (x'! e x'* == N e x'*))
----



0.  ~y*! e x'!

by lines [-1]
Next!"""
```

```
left()

"""

Line number 24



0.  (N e x*'? -> x'! e x*'?)

1.  (Ax'* : (x'! e x'* == N e x'*))
----



0.  x'! e x*'?

1.  ~y*! e x'!

by lines [-1]
Next!"""

back()
back()
getleft(1)

"""

Line number 23



0.  (N e x*'? -> x'! e x*'?)

1.  (Ax'* : (x'! e x'* == N e x'*))

2.  (x'! e x*'? -> N e x*'?)
----



0.  ~y*! e x'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 24



0.  (Ax'* : (x'! e x'* == N e x'*))

1.  (x'! e x*'? -> N e x*'?)
----



0.  N e x*'?

1.  ~y*! e x'!

by lines [-1]
Next!"""

back()
back()
back()
back()
left()

"""

Line number 22



0.  (x'! e x*'? == N e x*'?)

1.  (Ax'* : (x'! e x'* == N e x'*))
----
```

```
0.  ~y*! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 23




0.  (x'! e x*'? -> N e x*'?)

1.  (N e x*'? -> x'! e x*'?)

2.  (Ax'* : (x'! e x'* == N e x'*))
----



0.  ~y*! e x'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 23




0.  (N e x*'? -> x'! e x*'?)

1.  (Ax'* : (x'! e x'* == N e x'*))

2.  (x'! e x*'? -> N e x*'?)
----
```

```
0.  ~y*! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 24




0.  (Ax'* : (x'! e x'* == N e x'*))

1.  (x'! e x*'? -> N e x*'?)
----



0.  N e x*'?

1.  ~y*! e x'!

by lines [-1]
Next!"""

skip()

"""

Line number 25




0.  x'! e x*'?

1.  (Ax'* : (x'! e x'* == N e x'*))

2.  (x'! e x*'? -> N e x*'?)
----
```

```
0.  ~y*! e x'!

by lines [-1]
Next!"""

done()

"""

Line number 24




0.  (Ax'* : (x'! e x'* == N e x'*))

1.  (x'! e {x** | ~y*! e x**}
   -> N e {x** | ~y*! e x**})
----



0.  N e {x** | ~y*! e x**}

1.  ~y*! e x'!

by lines [-1]
Next!"""

right()

"""

Line number 26




0.  (Ax'* : (x'! e x'* == N e x'*))

1.  (x'! e {x** | ~y*! e x**}
   -> N e {x** | ~y*! e x**})
----
```

```
0.  ~y*! e N

1.  ~y*! e x'!

by lines [-1]
Next!"""

right()

"""

Line number 27




0.  y*! e N

1.  (Ax'* : (x'! e x'* == N e x'*))

2.  (x'! e {x** | ~y*! e x**}
   -> N e {x** | ~y*! e x**})
----



0.  ~y*! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 28




0.  y*! e {@x | ~@x = @x}

1.  (Ax'* : (x'! e x'* == N e x'*))

2.  (x'! e {x** | ~y*! e x**}
   -> N e {x** | ~y*! e x**})
```

```
----



0.  ~y*! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 29




0.  ~y*! = y*!

1.  (Ax'* : (x'! e x'* == N e x'*))

2.  (x'! e {x** | ~y*! e x**}
   -> N e {x** | ~y*! e x**})
----



0.  ~y*! e x'!

by lines [-1]
Next!"""

left()

"""

Line number 30




0.  (Ax'* : (x'! e x'* == N e x'*))

1.  (x'! e {x** | ~y*! e x**}
   -> N e {x** | ~y*! e x**})
```

```
----



0.  y*! = y*!

1.  ~y*! e x'!

by lines [-1]
Next!"""

done()

"""Done!"""

savetheorem('ZEN')
deft ("S", "{xEuEv&eua&~evu=x{wVewu=wv")
deff ("*I", "Ax>exae:axSa")
deft ("*N", "{nAi>&eZi:ai*Ieni")
start ("eZ*N")

"""

Line number 0



----



0.  Z e N*

by lines [-1]
Next!"""

right()

"""

Line number 1



----
```

```
0.  Z e {@n | (A@i : ((Z e @i
   & I*(a:@i))
   -> @n e @i))}

by lines [-1]
Next!"""

right()

"""

Line number 2



----



0.  (A@i : ((Z e @i
   & I*(a:@i))
   -> Z e @i))

by lines [-1]
Next!"""

right()

"""

Line number 3



----



0.  ((Z e i'!
   & I*(a:i'!))
   -> Z e i'!)

by lines [-1]
Next!"""
```

```
right()

"""

Line number 4



0.  (Z e i'!
   & I*(a:i'!))
----



0.  Z e i'!

by lines [-1]
Next!"""

left()

"""

Line number 5



0.  Z e i'!

1.  I*(a:i'!)
----



0.  Z e i'!

by lines [-1]
Next!"""

done()

"""Done!"""
```

```
savetheorem('PEANO1')
start ("Ax>ex*Ne:axS*N")

"""

Line number 0



----



0.  (Ax : (x e N*
   -> S(a:x) e N*))

by lines [-1]
Next!"""

right()

"""

Line number 1



----



0.  (x'! e N*
   -> S(a:x'!) e N*)

by lines [-1]
Next!"""

right()

"""

Line number 2
```

```
0.  x'! e N*
----



0.  S(a:x'!) e N*

by lines [-1]
Next!"""

right()

"""

Line number 3



0.  x'! e N*
----



0.  S(a:x'!)
   e {@n | (A@i : ((Z e @i
   & I*(a:@i))
   -> @n e @i))}

by lines [-1]
Next!"""

right()

"""

Line number 4



0.  x'! e N*
----
```

```
0.  (A@i : ((Z e @i
   & I*(a:@i))
   -> S(a:x'!) e @i))

by lines [-1]
Next!"""

right()

"""

Line number 5




0.  x'! e N*
----



0.  ((Z e i'!
   & I*(a:i'!))
   -> S(a:x'!) e i'!)

by lines [-1]
Next!"""

right()

"""

Line number 6




0.  (Z e i'!
   & I*(a:i'!))

1.  x'! e N*
----



0.  S(a:x'!) e i'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 7




0.  Z e i'!

1.  I*(a:i'!)

2.  x'! e N*
----



0.  S(a:x'!) e i'!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 7




0.  x'! e N*

1.  Z e i'!

2.  I*(a:i'!)
----



0.  S(a:x'!) e i'!
```

```
by lines [-1]
Next!"""

left()

"""

Line number 8




0.  x'! e {@n | (A@i : ((Z e @i
   & I*(a:@i))
   -> @n e @i))}

1.  Z e i'!

2.  I*(a:i'!)
----



0.  S(a:x'!) e i'!

by lines [-1]
Next!"""

left()

"""

Line number 9




0.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

1.  Z e i'!

2.  I*(a:i'!)
----
```

```
0.  S(a:x'!) e i'!

by lines [-1]
Next!"""

left()

"""

Line number 10




0.  ((Z e i*?
   & I*(a:i*?))
   -> x'! e i*?)

1.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

2.  Z e i'!

3.  I*(a:i'!)
----



0.  S(a:x'!) e i'!

by lines [-1]
Next!"""

left()

"""

Line number 11
```

```
0.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

1.  Z e i'!

2.  I*(a:i'!)
----



0.  (Z e i*?
   & I*(a:i*?))

1.  S(a:x'!) e i'!

by lines [-1]
Next!"""

right()

"""

Line number 13




0.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

1.  Z e i'!

2.  I*(a:i'!)
----



0.  Z e i*?

1.  S(a:x'!) e i'!

by lines [-1]
Next!"""
```

```
getleft(1)

"""

Line number 13




0.  Z e i'!

1.  I*(a:i'!)

2.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))
----



0.  Z e i*?

1.  S(a:x'!) e i'!

by lines [-1]
Next!"""

done()

"""

Line number 14




0.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

1.  Z e i'!

2.  I*(a:i'!)
----
```

```
0.  I*(a:i'!)

1.  S(a:x'!) e i'!

by lines [-1]
Next!"""

getleft(2)

"""

Line number 14




0.  I*(a:i'!)

1.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

2.  Z e i'!
----



0.  I*(a:i'!)

1.  S(a:x'!) e i'!

by lines [-1]
Next!"""

done()

"""

Line number 12




0.  x'! e i'!
```

```
1.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

2.  Z e i'!

3.  I*(a:i'!)
----



0.  S(a:x'!) e i'!

by lines [-1]
Next!"""

getleft(3)

"""

Line number 12



0.  I*(a:i'!)

1.  x'! e i'!

2.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

3.  Z e i'!
----



0.  S(a:x'!) e i'!

by lines [-1]
Next!"""

left()

"""
```

```
Line number 15



0.  (A@x : (@x e i'!
   -> S(a:@x) e i'!))

1.  x'! e i'!

2.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

3.  Z e i'!
----



0.  S(a:x'!) e i'!

by lines [-1]
Next!"""

left()

"""

Line number 16



0.  (x*? e i'!
   -> S(a:x*?) e i'!)

1.  (A@x : (@x e i'!
   -> S(a:@x) e i'!))

2.  x'! e i'!

3.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))
```

```
4.  Z e i'!
----



0.  S(a:x'!) e i'!

by lines [-1]
Next!"""

left()

"""

Line number 17




0.  (A@x : (@x e i'!
   -> S(a:@x) e i'!))

1.  x'! e i'!

2.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

3.  Z e i'!
----



0.  x*? e i'!

1.  S(a:x'!) e i'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 17
```

```
0.  x'! e i'!

1.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

2.  Z e i'!

3.  (A@x : (@x e i'!
   -> S(a:@x) e i'!))
----



0.  x*? e i'!

1.  S(a:x'!) e i'!

by lines [-1]
Next!"""

done()

"""

Line number 18



0.  S(a:x'!) e i'!

1.  (A@x : (@x e i'!
   -> S(a:@x) e i'!))

2.  x'! e i'!

3.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

4.  Z e i'!
----
```

```
0.  S(a:x'!) e i'!

by lines [-1]
Next!"""

done()

"""Done!"""

savetheorem('PEANO2')
start ("Aa>(& eZa :aa*I)(Ax > ex*N exa)")

"""

Line number 0



----



0.  (Aa : ((Z e a
   & I*(a:a))
   -> (Ax : (x e N* -> x e a))))

by lines [-1]
Next!"""

right()

"""

Line number 1



----



0.  ((Z e a'!
   & I*(a:a'!))
```

```
   -> (Ax : (x e N* -> x e a'!)))

by lines [-1]
Next!"""

right()

"""

Line number 2




0.  (Z e a'!
   & I*(a:a'!))
----



0.  (Ax : (x e N* -> x e a'!))

by lines [-1]
Next!"""

right()

"""

Line number 3




0.  (Z e a'!
   & I*(a:a'!))
----



0.  (x'! e N* -> x'! e a'!)

by lines [-1]
Next!"""

right()
```

```
"""

Line number 4



0.  x'! e N*

1.  (Z e a'!
   & I*(a:a'!))
----



0.  x'! e a'!

by lines [-1]
Next!"""

left()

"""

Line number 5



0.  x'! e {@n | (A@i : ((Z e @i
   & I*(a:@i))
   -> @n e @i))}

1.  (Z e a'!
   & I*(a:a'!))
----



0.  x'! e a'!

by lines [-1]
Next!"""

left()
```

```
"""

Line number 6



0.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

1.  (Z e a'!
   & I*(a:a'!))
----



0.  x'! e a'!

by lines [-1]
Next!"""

left()

"""

Line number 7



0.  ((Z e i'?
   & I*(a:i'?))
   -> x'! e i'?)

1.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

2.  (Z e a'!
   & I*(a:a'!))
----
```

```
0.  x'! e a'!

by lines [-1]
Next!"""

left()

"""

Line number 8
```

```
0.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

1.  (Z e a'!
   & I*(a:a'!))
----
```

```
0.  (Z e i'?
   & I*(a:i'?))

1.  x'! e a'!

by lines [-1]
Next!"""

getleft(1)

"""

Line number 8
```

```
0.  (Z e a'!
   & I*(a:a'!))

1.  (A@i : ((Z e @i
   & I*(a:@i))
```

```
   -> x'! e @i))
----



0.  (Z e i'?
   & I*(a:i'?))

1.  x'! e a'!

by lines [-1]
Next!"""

left()

"""

Line number 10



0.  Z e a'!

1.  I*(a:a'!)

2.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))
----



0.  (Z e i'?
   & I*(a:i'?))

1.  x'! e a'!

by lines [-1]
Next!"""

right()

"""

Line number 11
```

```
0.  Z e a'!

1.  I*(a:a'!)

2.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))
----



0.  Z e i'?

1.  x'! e a'!

by lines [-1]
Next!"""

done()

"""

Line number 12




0.  Z e a'!

1.  I*(a:a'!)

2.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))
----



0.  I*(a:a'!)

1.  x'! e a'!
```

```
by lines [-1]
Next!"""

getleft(1)

"""

Line number 12




0.  I*(a:a'!)

1.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

2.  Z e a'!
----



0.  I*(a:a'!)

1.  x'! e a'!

by lines [-1]
Next!"""

done()

"""

Line number 9




0.  x'! e a'!

1.  (A@i : ((Z e @i
   & I*(a:@i))
   -> x'! e @i))

2.  (Z e a'!
```

```
   & I*(a:a'!))
----



0.  x'! e a'!

by lines [-1]
Next!"""

done()

"""Done!"""

savetheorem('INDUCTION')
start ("Ax~=Z:axS")

"""

Line number 0



----



0.  (Ax : ~Z = S(a:x))

by lines [-1]
Next!"""

right()

"""

Line number 1



----



0.  ~Z = S(a:x'!)
```

```
by lines [-1]
Next!"""

right()

"""

Line number 2



0.  Z = S(a:x'!)
----



by lines [-1]
Next!"""

left()

"""

Line number 3



0.  (Ax* : (Z e x*
   == S(a:x'!) e x*))
----



by lines [-1]
Next!"""

left()

"""

Line number 4
```

```
0.  (Z e x''?
   == S(a:x'!) e x''?)

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))
----



by lines [-1]
Next!"""

setunknown ("x''","{uAx>exuEyeyx")

"""

Line number 4




0.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   == S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))
----



by lines [-1]
Next!"""

left()

"""

Line number 5




0.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
```

```
   e {u | (Ax : (x e u -> (Ey : y e x)))})

1.  (S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> Z e {u | (Ax : (x e u -> (Ey : y e x)))})

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))
----



by lines [-1]
Next!"""

getleft(1)

"""

Line number 5




0.  (S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> Z e {u | (Ax : (x e u -> (Ey : y e x)))})

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



by lines [-1]
Next!"""

left()

"""

Line number 6
```

```
0.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

1.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))}

by lines [-1]
Next!"""

right()

"""

Line number 8



0.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

1.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  (Ax : (x e S(a:x'!)
   -> (Ey : y e x)))

by lines [-1]
Next!"""

right()
```

```
"""

Line number 9




0.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

1.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  (x'*! e S(a:x'!)
   -> (Ey : y e x'*!))

by lines [-1]
Next!"""

right()

"""

Line number 10




0.  x'*! e S(a:x'!)

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  (Ey : y e x'*!)
```

```
by lines [-1]
Next!"""

left()

"""

Line number 11




0.  x'*! e {@x | (E@u : (E@v : (@u e x'!
   & (~@v e @u
   & @x = {@w | (@w e @u V @w = @v)})))}

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  (Ey : y e x'*!)

by lines [-1]
Next!"""

left()

"""

Line number 12




0.  (E@u : (E@v : (@u e x'!
   & (~@v e @u
   & x'*! = {@w | (@w e @u V @w = @v)})))

1.  (Ax* : (Z e x*
```

```
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  (Ey : y e x'*!)

by lines [-1]
Next!"""

left()

"""

Line number 13



0.  (E@v : (u'! e x'!
   & (~@v e u'!
   & x'*! = {@w | (@w e u'! V @w = @v)})))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  (Ey : y e x'*!)

by lines [-1]
Next!"""

left()

"""
```

```
Line number 14



0.  (u'! e x'!
   & (~v'! e u'!
   & x'*! = {@w | (@w e u'! V @w = v'!)}))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  (Ey : y e x'*!)

by lines [-1]
Next!"""

right()

"""

Line number 15



0.  (u'! e x'!
   & (~v'! e u'!
   & x'*! = {@w | (@w e u'! V @w = v'!)}))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----
```

```
0.  y'? e x'*!

1.  (Ey : y e x'*!)

by lines [-1]
Next!"""

setunknown ("y'","v")

"""

Line number 15




0.  (u'! e x'!
   & (~v'! e u'!
   & x'*! = {@w | (@w e u'! V @w = v'!)}))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  v e x'*!

1.  (Ey : y e x'*!)

by lines [-1]
Next!"""

left()

"""

Line number 16
```

```
0.  u'! e x'!

1.  (~v'! e u'!
   & x'*! = {@w | (@w e u'! V @w = v'!)})

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  v e x'*!

1.  (Ey : y e x'*!)

by lines [-1]
Next!"""

getleft(1)

"""

Line number 16




0.  (~v'! e u'!
   & x'*! = {@w | (@w e u'! V @w = v'!)})

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

3.  u'! e x'!
----
```

```
0.  v e x'*!

1.  (Ey : y e x'*!)

by lines [-1]
Next!"""

left()

"""

Line number 17




0.  ~v'! e u'!

1.  x'*! = {@w | (@w e u'! V @w = v'!)}

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

4.  u'! e x'!
----



0.  v e x'*!

1.  (Ey : y e x'*!)

by lines [-1]
Next!"""

getleft(1)

"""

Line number 17
```

```
0.  x'*! = {@w | (@w e u'! V @w = v'!)}

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

3.  u'! e x'!

4.  ~v'! e u'!
----



0.  v e x'*!

1.  (Ey : y e x'*!)

by lines [-1]
Next!"""

varelim()

"""

Line number 18



0.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

1.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

2.  u'! e x'!

3.  ~v'! e u'!
----
```

```
0.  v e {@w | (@w e u'! V @w = v'!)}

1.  (Ey : y e {@w | (@w e u'! V @w = v'!)})

by lines [-1]
Next!"""

right()

"""

Line number 19




0.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

1.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

2.  u'! e x'!

3.  ~v'! e u'!
----



0.  (v e u'! V v = v'!)

1.  (Ey : y e {@w | (@w e u'! V @w = v'!)})

by lines [-1]
Next!"""

right()

"""

Line number 20
```

```
0.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

1.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

2.  u'! e x'!

3.  ~v'! e u'!
----



0.  v e u'!

1.  v = v'!

2.  (Ey : y e {@w | (@w e u'! V @w = v'!)})

by lines [-1]
Next!"""

back()
back()
back()
back()
back()
back()
back()
back()
back()
setunknown ("y'","'v")

"""

Line number 15



0.  (u'! e x'!
   & (~v'! e u'!
```

```
   & x'*! = {@w | (@w e u'! V @w = v'!)}))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  v'! e x'*!

1.  (Ey : y e x'*!)

by lines [-1]
Next!"""

left()

"""

Line number 16




0.  u'! e x'!

1.  (~v'! e u'!
   & x'*! = {@w | (@w e u'! V @w = v'!)})

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  v'! e x'*!

1.  (Ey : y e x'*!)
```

```
by lines [-1]
Next!"""

getleft(1)

"""

Line number 16




0.  (~v'! e u'!
   & x'*! = {@w | (@w e u'! V @w = v'!)})

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

3.  u'! e x'!
----



0.  v'! e x'*!

1.  (Ey : y e x'*!)

by lines [-1]
Next!"""

left()

"""

Line number 17




0.  ~v'! e u'!
```

```
1.  x'*! = {@w | (@w e u'! V @w = v'!)}

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

4.  u'! e x'!
----



0.  v'! e x'*!

1.  (Ey : y e x'*!)

by lines [-1]
Next!"""

getleft(1)

"""

Line number 17



0.  x'*! = {@w | (@w e u'! V @w = v'!)}

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

3.  u'! e x'!

4.  ~v'! e u'!
----
```

```
0.  v'! e x'*!

1.  (Ey : y e x'*!)

by lines [-1]
Next!"""

varelim()

"""

Line number 18




0.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

1.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

2.  u'! e x'!

3.  ~v'! e u'!
----



0.  v'! e {@w | (@w e u'! V @w = v'!)}

1.  (Ey : y e {@w | (@w e u'! V @w = v'!)})

by lines [-1]
Next!"""

right()

"""

Line number 19
```

```
0.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

1.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

2.  u'! e x'!

3.  ~v'! e u'!
----



0.  (v'! e u'! V v'! = v'!)

1.  (Ey : y e {@w | (@w e u'! V @w = v'!)})

by lines [-1]
Next!"""

right()

"""

Line number 20




0.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

1.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

2.  u'! e x'!

3.  ~v'! e u'!
----



0.  v'! e u'!
```

```
1.  v'! = v'!

2.  (Ey : y e {@w | (@w e u'! V @w = v'!)})

by lines [-1]
Next!"""

getleft(1)

"""

Line number 20



0.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

1.  u'! e x'!

2.  ~v'! e u'!

3.  (Ax* : (Z e x*
   == S(a:x'!) e x*))
----



0.  v'! e u'!

1.  v'! = v'!

2.  (Ey : y e {@w | (@w e u'! V @w = v'!)})

by lines [-1]
Next!"""

getright(1)

"""

Line number 20
```

```
0.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})

1.  u'! e x'!

2.  ~v'! e u'!

3.  (Ax* : (Z e x*
   == S(a:x'!) e x*))
----



0.  v'! = v'!

1.  (Ey : y e {@w | (@w e u'! V @w = v'!)})

2.  v'! e u'!

by lines [-1]
Next!"""

done()

"""

Line number 7




0.  Z e {u | (Ax : (x e u -> (Ey : y e x)))}

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----
```

```
by lines [-1]
Next!"""

left()

"""

Line number 21




0.  (Ax : (x e Z -> (Ey : y e x)))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



by lines [-1]
Next!"""

left()

"""

Line number 22




0.  (x*'? e Z -> (Ey : y e x*'?))

1.  (Ax : (x e Z -> (Ey : y e x)))

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
```

```
----



by lines [-1]
Next!"""

left()

"""

Line number 23




0.  (Ax : (x e Z -> (Ey : y e x)))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  x*'? e Z

by lines [-1]
Next!"""

right()

"""

Line number 25




0.  (Ax : (x e Z -> (Ey : y e x)))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))
```

```
2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  x*'? e {@x | (A@y : ~@y e @x)}

by lines [-1]
Next!"""

right()

"""

Line number 26




0.  (Ax : (x e Z -> (Ey : y e x)))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  (A@y : ~@y e x*'?)

by lines [-1]
Next!"""

right()

"""

Line number 27
```

```
0.  (Ax : (x e Z -> (Ey : y e x)))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  ~y*! e x*'?

by lines [-1]
Next!"""

right()

"""

Line number 28



0.  y*! e x*'?

1.  (Ax : (x e Z -> (Ey : y e x)))

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



by lines [-1]
Next!"""

setunknown ("x*'","N")
```

```
"""

Line number 28



0.  y*! e N

1.  (Ax : (x e Z -> (Ey : y e x)))

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



by lines [-1]
Next!"""

left()

"""

Line number 29



0.  y*! e {@x | ~@x = @x}

1.  (Ax : (x e Z -> (Ey : y e x)))

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----
```

```
by lines [-1]
Next!"""

left()

"""

Line number 30




0.  ~y*! = y*!

1.  (Ax : (x e Z -> (Ey : y e x)))

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



by lines [-1]
Next!"""

left()

"""

Line number 31




0.  (Ax : (x e Z -> (Ey : y e x)))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
```

```
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  y*! = y*!

by lines [-1]
Next!"""

done()

"""

Line number 24



0.  (Ey : y e N)

1.  (Ax : (x e Z -> (Ey : y e x)))

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



by lines [-1]
Next!"""

left()

"""

Line number 32
```

```
0.  y''! e N

1.  (Ax : (x e Z -> (Ey : y e x)))

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



by lines [-1]
Next!"""

left()

"""

Line number 33




0.  y''! e {@x | ~@x = @x}

1.  (Ax : (x e Z -> (Ey : y e x)))

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



by lines [-1]
Next!"""

left()

"""
```

```
Line number 34



0.  ~y''! = y''!

1.  (Ax : (x e Z -> (Ey : y e x)))

2.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

3.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



by lines [-1]
Next!"""

left()

"""

Line number 35




0.  (Ax : (x e Z -> (Ey : y e x)))

1.  (Ax* : (Z e x*
   == S(a:x'!) e x*))

2.  (Z e {u | (Ax : (x e u -> (Ey : y e x)))}
   -> S(a:x'!)
   e {u | (Ax : (x e u -> (Ey : y e x)))})
----



0.  y''! = y''!
```

```
by lines [-1]
Next!"""

done()

"""Done!"""

savetheorem('PEANO3')
```