Documentation for the Python version of the Marcel Theorem Prover

M. Randall Holmes

September 9, 2021

1 Introduction

The Python version underwent major updates in November 2015.

This is not quite the same prover as Marcel, as implemented in Standard ML elsewhere. Like the Marcel program implemented in ML, it is an implementation of NFU in sequent calculus. The original Marcel program was inspired by a paper of Marcel Crabbé giving a semantic proof of cut elimination for SF (stratified comprehension with no extensionality at all); this paper has some further motivation in contemplation of the logic of Frege with the stratification criterion imposed to avoid the paradoxes.

This section is technical, addressed to someone with knowledge of the kind of set theory being implemented. A naive user should skip it and go to the next section.

This version does a better job than the ML version of demonstrating just how simple NFU is to implement. The original ML version checked stratification itself: here we require that bound variables have type indices, which makes the typing algorithm quite simple. The original ML version used a fancy algorithm for substitution which automatically renamed bound variables to avoid collisions (so that variables in the original Marcel had numbers attached anyway, but indices rather than types): here we simply avoid ever substituting an expression with a free variable in it into another expression, so substitution is extremely simple. Further, the term structure here is much simpler. Quantifications are treated as applications of unary operators to sets, and in general kinds of terms are reduced to a minimum. The type systems for propositional and object terms are unified. This system

has sets and functions as cognate primitive notions from the outset. So far the type level ordered pair which one surely would eventually want to introduce is no part of the logic, though I think I see where it may be introduced naturally if the relationship between sets and functions is clarified. There is a built in definite description operator as part of the logic (important for Frege's system, and also important for showing that functional relations always have analogous functions). The Hilbert symbol could be introduced in the same way.

In November 2015, I added the ability to declare operators with strongly cantorian range, and to recognize when defined operators have strongly cantorian range. This is useful because the axiom of counting (the assertion that the natural numbers make up a strongly cantorian set) is required for the implementation of Frege's system. In general, it is useful because it gives the system more mathematical power.

The prover implements a higher order logic based on NFU. In fact, the logic is stratified comprehension with weak extensionality implemented via the presence of abstracts as a primitive notion: $\{x \mid \phi\}$ is equal to $\{x \mid \psi\}$ iff $(\forall x.\phi \leftrightarrow \psi)$, and nothing is said about extensions of non-sets (objects u such that $u \neq \{x \mid x \in u\}$) at all. There are similar axioms relating function abstraction and function application.

The original flavor of Marcel incorporates an algorithm which computes stratifications. This prover does not have this feature. In the language of Python Marcel, variables have type. Object expressions with free variables have type. Object expressions with no free variables (constants) have no type assigned to them. In any atomic formula $T = U$, if both x and y have type their types must be the same. In an atomic formula $T \in U$, if both x and y have type, the type of y is one higher than the type of x. Every binary predicate in the language has a similar typing rule (a fixed displacement between the types of the arguments is enforced, if both have assigned type). The highest arity of a predicate in our language is 2. A set $\{x \mid \phi\}$ is lassigned type one higher than that of x if it contains a free variable (occurrences of x are bound in this notation of course). There are function terms which are handled similarly (the type differential is also 1) There are unary and binary operations on objects: each of them has a fixed displacement between types of the output and the two inputs which are enforced if an input is assigned type; of course if neither input is assigned type, the whole term is not assigned type.

The strongly cantorian range update has the following effects: a term

whose main operator has strongly cantorian output type is not assigned type (the displacement between the types of its arguments, if it has two, is enforced as usual). A term in which a unary operator with s.c. output type is applied to a variable is referred to as a strongly cantorian variable. An abstract (set or function) in which the body of the abstraction is not assigned type and all occurrences of the main bound variable are included in a strongly cantorian variable is not assigned type (it belongs either to the power set of a given s.c. set or the function space with a given s.c. domain and a given s.c. range). This type deduction is strong enough to allow defined operators to be assigned s.c. output type automatically where appropriate. Primitive operators can be declared to have s.c. output: this is how the axiom of counting is imposed (a primitive operation is declared with s.c. output and identified by axiom with a retraction from the universe onto the natural numbers).

Rules for substitution in this language are very simple, as we never substitute anything which contains a free variable for a variable or other atomic object term. We do have a semantics in mind for terms with free variables (whether object or proposition): we view them as second order objects, "concepts" in Frege's sense. In the original version of Marcel, there is a quite complex substitution algorithm involving introduction of fresh variables automatically: it has been instructive to see that this is not needed. I have not even implemented alpha-equivalence of sets and functions, though I surely will eventually: I am not sure that such a function will see enormous use in practice.

The restriction to binary predicates and operations is harmless for the usual reasons: the Kuratowski ordered pair is definable in NFU. So far I have not given a type level ordered pair a fundamental logical role in this version of the prover. There are complex built-in optimizations of the use of ordered pairs and projections in the original, which are not present here as yet.

There is no use of matching in the current version. In general, the core of this version is extremely simple: I do not introduce complex optimizations which might cause logical bugs. This is a virtue. That doesn't mean that some of the things which are not present so far might not make it in. Currently one uses theorems by being able to freely add their top level propositions as new assumptions, which means that a theorem with deep logical structure takes some few steps to apply. The matching approach to using theorems, which allows use of nontrival sequents as in effect derived rules,

might eventually be wanted. Being able to extract proved sequents from a proof (even the current one) other than at the top and record them as theorems might be desirable. The dazzling effects in the ML version of Marcel caused by allowing matching to force instantiations might be wanted.

The logic of functions which is now installed is interesting in that it is parallel to but independent of the logic of sets. The presence of functions which are one type higher than their arguments and values does not imply Infinity: the assumption that all functions are sets (in the presence of definite descriptions) is enough to imply this, but this is not so far part of the logic.

2 The Language and Type System of the Prover

The prover has a moderately complex language, in which each term is assigned a type. There are really only two kinds of term, proposition terms (statements) and object terms (names of things). When terms contain free (object) variables we think of them as representing abstractions from statements (sets or relations) or abstractions from objects (functions). A proposition is assigned type `prop` in all cases. An object term which contains no free variables is assigned type `constant`, while an object term containing free variables will be assigned an integer type. Although there is really only one kind of object, it is useful to think of type 0 as an unspecified type of individuals, type 1 as the type of sets of type 0 objects or functions from type 0 objects to type 0 objects (plus anonymous other objects), type 2 as the type of sets of type 1 objects or functions from type 1 objects to type 1 objects (plus anonymous other objects), and so forth: type $i+1$ terms are to be thought of as representing sets of type i objects or functions from type i objects to type i objects (plus anonymous other objects). The reader sophisticated enough to worry about more complex function types should notice that an object of type i can be coded in type $i+1$ by its singleton set, and this can be iterated, and so heterogeneous function types can be coded into our flat function types.

Atomic terms of the language are

propositional variables: strings of lower case letters followed by a question mark `?`, which are assigned type `prop`

instantiables: (object terms which can be freely assigned new values in a proof), which are strings of lower case letters followed by a dollar sign

`$` followed by an optional string of digits.

constants: strings of digits (numerals) or strings of lower case letters followed by an optional index (a string of digits prefixed with an underscore _), which are typed `constant`. A constant without an index is only well-typed if it is declared or defined; indexed constants may be used freely. A single lower case letter is not a constant but a variable.

variables: (strictly speaking, object variables) strings of lower case letters followed by strings of digits, which are assigned type equal to the nonnegative integer value of the string of digits understood as a numeral. A single letter is read as a type zero variable.

There is one further class of identifiers, and further characters used in syntax.

operators: An operator term is a string of uppercase letters followed by an optional index or a string of special characters (basically, those typewriter characters other than quotes and backslash which are not otherwise in use) followed by an optional index. An operator term cannot be used unless its nonindexed form has been assigned a type signature: once this is done, it can be used bare or with any index.

Operators beginning with `:` are reserved as abstractors.

other characters: paired forms (parentheses, braces and brackets) and the dot `..` The parser completely ignores whitespace and all other characters not used in Marcel terms, even in the middle of identifiers!

We now discuss term constructions and associated type rules.

type signatures A type signature is a sequence of two or three items each of which is either a nonnegative integer, `sc`, or `prop`. The first term is the intended type of the whole term built with an operator with that type signature. The second term is the intended type of the term to the left of a binary operator or the term following a unary (always prefix) operator with that type signature. The third term is the intended type of the term following a binary operator with that type signature.

unary terms: An operator followed by a term is a term with type `prop` if its type signature starts with `prop` and the type of the term matches the type of the second term of the type signature of the operator (where only `prop` matches `prop` and any nonnegative integer or the type `constant` match an integer. In the latter case, the type of the new prefix operator term is `constant` if the type of the component term is `constant` or if the output type of the operator is `sc` and otherwise has the same difference from the first component of the type signature as the difference between the type of the component term and the second component of the type signature.

binary terms A first component term followed by an operator (not beginning with `:`) followed by a second component term is a term iff the types of the first and second component terms match the second and third components of the type signature. The only ways that a pair of types of component terms can fail to match a pair of types from a type signature is that `prop` appears in one pair without a matching `prop` in the other pair, or that both pairs are pairs of integers and they have unequal differences. The type of the whole term will be `prop` if the first term of the type signature is `prop`, will be `constant` iff the types of both the component terms are `constant` or if the output type is `sc`, and otherwise will have integer type differing from the integer type of either of its components by the same amount that the first component of the operator type signature differs from the appropriate component of the type signature corresponding to that component term.

abstraction terms: Currently there are only two abstraction operators `:` and `:>`. Each of these expects a variable to the left and a term to the right which will be of type `prop` for `:` and of type `constant` or an integer type for `:>`. The type of a term of either of these forms is one higher than the type of the variable if the whole term contains any free variables (see the next item), and otherwise is `constant`. It will be `constant` as well if all occurrences of the variable to the left of the abstractor in the term on the right of the abstractor are in s.c. variables and the type of the term on the right of the abstractor is `constant`.

free and bound variables: A variable is free in itself. It is not free in any atomic term. It is free in a unary or binary term which is not an abstraction term iff if it is free in one of the component subterms. It is

free in an abstraction term iff it is free in the right subterm and different from the left subterm. If complex abstraction operators `::` and `::>` are introduced which can take complex left subterms, the definition of free variables (and the definition of substitution for a variable) will become more complex. It should be noted that no term entered into a proof by the user will contain any free variable.

paired forms and operator precedence: A term enclosed in parentheses, braces, or brackets is a term of the same type. Marcel will display parentheses around propositions, braces around abstractions, and brackets around other object terms, but this is not required in input. Operators have precedence. Even precedences group to the right and odd precedences group to the left. Marcel displays only those parentheses that are required.

separating adjacent operators: A dot `.` may be used to separate operators when they are adjacent (which only happens when a unary or binary operator is immediately followed by a unary operator. The parser and print functions have been modified so that the dot is optional (Marcel will turn whitespace between operators into a dot internally, the only situation where the parser pays any attention to whitespace) and is always expressed as a space in the display. We might possibly restore the compact display function, which would reintroduce the dot.

Some operators are predeclared.

abstractors: The infixes `:` and `:>`, both with precedence 0. The first forms sets, the second forms functions.

negation: The unary operator $\sim$ with type signature `[prop,prop]`. This binds more tightly than the other logical operations.

binary propositional connectives: `&, V, ->, ==`, the conjunction, disjunction, implication and biconditional operations, each with signature `[prop,prop,prop]`, listed in decreasing order of precedence. The first two group to the left, the others group to the right.

binders: The universal quantifier `A` and the existential quantifier `E`. These have precedence 0. They are understood as unary operators which are applied to abstraction terms (sets) in their usual contexts, so they have signature `[prop,0]`.

primitive logical relations: `=` is equality, `IN` is membership (the overloading of `E` in the original version of Marcel is not supported by the much simpler (and cleaner) parser here). The signature of `=` is `[prop,0,0]` and the signature of `IN` is `[prop,0,1]`. All predicates and relations have the same precedence, binding more tightly than the logical connectives, unless the user meddles (the binders are in a sense unary predicates but they are assigned lowest precedence for reasons of familiar appearance of syntax).

predeclared operations: `‘` is function application. It has signature `[0,1,0]`, odd precedence (grouping to the left) higher than that of other operations. `THE` is the definite description operator. It acts on a set and returns its sole element if it is a singleton (its behavior otherwise is unspecified). It has lowest precedence (being a binder like `A` and `E`).

3 The Logic

The Marcel prover manipulates sequents. A sequent is a pair of lists of statements $\Gamma \vdash \Delta$. A sequent is said to be valid iff any possible interpretation which makes all the sentences in Γ true makes some sentence in Δ true. In the usual sequent notation, the assumptions are to the left of the turnstile $\vdash$ and the conclusions are to the right: though in Marcel's notation the assumptions are above a line and the conclusions are below a line, we still refer to assumptions as "left" and conclusions as "right".

A proof in Marcel is a sequent paired with either the object `goal` (indicating that the sequent has not been proved to be valid) or a list of proofs completion of which will establish the validity of the sequent. The hidden machinery of Marcel rotates the subproofs inside a proof until a sequent paired with `goal` is leftmost, and displays that sequent for attention. The `Nextgoal()` command will rotate through all the goals currently available (in a not very efficient way, which can repeat goals before covering all of them). When a proof is complete the prover displays `Q. E. D.`

One can view the entire structure of the current proof at any time in a format readable by a very patient human using `Showall()`.

The proof begins with the command `Start(P)`, which creates a goal with `|- P` as the sequent. `P` will contain no free variables.

The `Left()` command acts on the first assumption in the sequent. The

`Right()` command acts on the first conclusion in the sequent. The ways in which they act are best explored by trying them out! They treat logical connectives in familiar ways. In certain cases (for example, conjunctions to be proved, disjunctions assumed, biconditionals to be proved) a single sequent will generate two sequents to be proved. In such a case, the prover just keeps the next sequent(s) to be proved in the background and serves them up when you are finished with a previous goal. For universal goals or existential assumptions these commands introduce general objects about which no additional assumptions are made (these are implemented as constants with indices high enough that they are new in the context). For existential conclusions or universal assumptions, instantiables are introduced, with high enough indices to be new, with the further property that they can be globally replaced at any time with a term containing only subterms strictly "older" (as measured by index) than the instantiable. This means that you can delay decision as to what the witness to an existential goal is, or what object you will plug into a universal hypothesis, until the formal development of the proof gives you some ideas. The command `Inst(t,v)` replaces the instantiable `v` with the term `t` (which will not contain free variables).

Commands `PropInst` and `OpInst` will make global substitutions of a bound-variable-free sentence for a propositional variable or an operator for an indexed operator, respectively. These commands support the actual use of proofs of propositional logic, or general proofs about relations using operators. Subsitution of a general abstraction (similar to the input for the `Define` command) for an indexed operator might eventually be supported.

The `Getleft()`, `Getleft2()`, `Getright()`, `Getright2()` commands rotate the assumption or conclusion lists (one version for each direction of rotation). This is needed to bring terms into the first position in either list, where the `Left()` and `Right()` commands can act on them.

Equations $x = y$ get expanded to $(\forall z.x \in z \leftrightarrow y \in z)$, except in the case of a conclusion of the form $\{x \mid \phi\} = \{x \mid \psi\}$, which expands to $(\forall x.\phi \leftrightarrow \psi)$ [and similarly for functions, not yet implemented]. Equations $a \in \{x \mid \phi(x)\}$ are expanded to $\phi(a)$ as expected.

Definitions are expanded when no other logical move works (but definition expansion is preferred to the default right rule for equality).

Proved sequents in the current proof can be saved (`Savetheorem(label,name)`; the entire theorem can be saved once `Q. E. D.` is displayed) and are used not in the complex way involving matching implemented in the ML version of Marcel, but essentially by introducing a cut. The command is `Usethm(name)`.

Unary and binary operators can be defined using a rather complex syntax: the argument of `Define` is a list consisting of the name of the operator, followed by one or two variables then the body of the operator. Well-formedness is determined by typing the body, and the type signature is determined from the types of the variable(s) and the type of the body in the obvious way. The variables are then replaced with constants `.arg_1` and `.arg_2` which cannot be entered by the user and so cannot be confused with anything in the defined operation. Definitional expansion is a matter of substitution. Notice that defined operators can be heterogeneous: they do not have to represent set functions. Constants can also be defined by the `Defineconstant` command, and primitive constants can be declared with the `Declareconstant` command (as primitive operators can be declared using `declareunary` and `declarebinary`). The body of a definition may not contain a propositional variable or any indexed term.

Theorems and definitions can be viewed using `Showthm` and `Showdef`.

For those who are used to having just one conclusion in a logical argument, the `Oneconclusion()` command modifies the display so that conclusions after the first are negated and starred and appended to the list of assumptions. `Left()` or `Getleft()` will not touch these starred assumptions; `Getright()` will have its usual effect on them, but it will look like a contrapositive maneuver. `Manyconclusions()` restores the usual behaviour. This is different from the similar device in Marcel, which actually changes the sequent and so inessentially changes the logic; this is purely a change in the display, but it may have the same value in giving the arguments a more familiar form.

4 Command Reference

This lists all the user commands in the order in which they appear in the source.

It is a useful point that there are no commands which take indices of terms in sequents as arguments. This may possibly improve script portability, I am not certain. All arguments are terms or types.

setprec: a string argument followed by an integer argument. Set the precedence of the operator named by the string to the integer.

setprecsame: two string arguments: assign precedence to the operator

named by the first string equal to that already assigned to the second string.

setprec{even/odd}{above/below}: Four different commands, each with two string arguments. Assign the operator named by the first string an even or odd precedence just above or just below the precedence of the second argument, raising all precedences above the cut point by two. This command also moves the thresholds for minimum propositional connective, predicate/relation, and operation precedences (which guide the declarunary and declarebinary commands in setting precedences). The precedence system in this version of Marcel is more general than that in the ML version, which treated propositions and objects in different frameworks.

declareunary: one string argument followed by two type arguments. This sets the type signature of the operator named by the string to the list of the second and third arguments. It also makes an initial assignment of precedence to the operator which the user may want to adjust. This declares unary prefix operators.

declarebinary: one string argument followed by three type arguments. As above, except that it sets the type signature to the list of the second third and fourth arguments. This declares binary infix operators.

Declareconstant: This takes one string argument, the name of a constant to be declared.

Oneconclusion(): Modifies the display so that second and further conclusions appear negated and starred at the end of the assumption list. This is purely a modification of the display. It affects the output of Showall() as well as the current sequent display.

Manyconclusions(): Restores the display of multiple conclusions.

Start: takes one string argument, which should parse as a theorem to be proved. Creates the initial goal of a proof, a sequent with nothing on the left and the proposition expressed by the string argument on the right.

Nextgoal(): changes the goal displayed in an odd way which is guaranteed eventually to visit all unproved goals in a proof, though it may repeat some more than once in a full cycle.

Look(): Display the current goal to be manipulated.

Done(): This command will mark the current goal as complete if the first assumption and first conclusion are the same.

Left(): Transform the current goal in a way dictated by the logical form of the first assumption.

Right(): Transform the current goal in a way dictated by the logical form of the first conclusion.

Pruneleft(): Drop the first assumption in the current goal.

Pruneright(): Drop the first conclusion in the current goal.

Define: Takes a single list with three (unary case) or four (binary case) entries as an argument. The first term in the list is the name of the operator to be defined. The last term is the term to which an instance of this operation is to be expanded, with arguments replacing the variables which appear as the second and third terms (no other variables should appear free). The type signature of the defined operator is deduced from the given data. The body of a definition may not contain any indexed term or operator, nor may it contain a propositional variable.

Defineconstant: Takes two string arguments, the name of a constant and the term to which it is to be expanded (which should contain no free variables). The body of a definition may not contain any indexed term or operator, nor may it contain a propositional variable.

Getleft(), Getleft2(): Rotate the order of the assumptions in the current goal.

Getright(), Getright2(): Rotate the order of the conclusions in the current goal.

GetleftE(), Getleft2E(): Rotate the order of the assumptions other than the first one. Used to support rewriting on the left.

Cut: One string argument, to be read as a proposition. Converts the current goal into two, one with the proposition argument as an additional conclusion, and one with it an additional hypothesis. Effectively, the argument is a lemma introduced to simplify the proof.

Converse(): Sets the direction of rewriting using an equation hypothesis to right to left ($a = b$ rewrites b to a)

Direct(): Sets the direction of rewriting using an equation hypothesis to left to right ($a = b$ rewrites a to b)

Inleft(): Rewriting using an equational hypothesis will take place in the second assumption.

Inright(): Rewriting using an equational hypothesis will take place in the first conclusion.

Equal0(), Equal1(), Equal2(): Each of these commands causes an equational hypothesis to be used for rewriting in a particular direction in a particular target. See previous commands. The equation used is the first assumption. Equal0 does a complete rewrite, Equal1 carries out the leftmost single rewrite, Equal1 carries out the rightmost single rewrite. Neither Equal1 nor Equal2 will rewrite an occurrence of the source term which is a subterm of an instance of the target term. Use the four previous commands to set up the direction and target of the rewriting, and use GetleftE, Getleft2E to position the target if it is an assumption.

Equaldr0(),Equaldl0(),Equalcr0(),Equalcl0(), etc. These commands combine the correct Direct/Converse and the correct Inright/Inleft command with the correct numerically labelled Equal command.

Inst: This command takes two string arguments, the first an object term with no free variables and the second an instantiable with index higher than any appearing in the term. The index of a non-indexed instantiable is planned to be taken as infinite (not yet implemented as I write this). The term is to be substituted for the instantiable globally throughout the current proof.

PropInst: Two string arguments, a bound-variable-free proposition and a propositional variable. The sentence is to replace the variable throughout the current proof. This is intended to support use of theorems of propositional logic.

OpInst: Two string operators, the first an operator, the second an indexed operator. The first is to replace the second throughout the current proof.

Showall(): Display a human readable if vast and boring readout of the current proof.

Saveproof: One string argument. Saves the current proof with the argument as key.

Savetheorem: Two string arguments. Saves the sequent in the current proof labelled by the first argument to the theorem list with the second argument as key, if the proof is complete.

Axiom: Two string arguments, the name and text of a proposition which is asserted as a theorem. The body of an axiom may not contain any indexed term or operator, nor may it contain a propositional variable.

Loadproof: One string argument. Sets the current proof to the saved proof named by the argument.

Usethm: One string argument. Uses the theorem named by the argument in essence as a rule: new goals prove its hypotheses and add its conclusions as new assumptions, with general variables in the theorem converted to instantiables so their values can be set as desired. (This is the function Thmcut of the ML version).

Showdef: One string argument. Displays the definition of the concept (constant or operator) denoted by the string.

Showthm: One string argument. Displays the theorem named by the argument.

5 The Source Code

Here is the source code in Python, mod its tendency to sail off past the right margin.

```
# Python version of Marcel -- 10/15/2015 7:30 PM desktop home

# All rights reserved by M. Randall Holmes, the author, 2014.  You are allowed
# to use the file and even modify it for any noncommercial purpose, but this
# notice  must be preserved.

# version of 11/19/2015, narrative comments on propositional and
# quantifier proof steps.
# Declarations of operators and constants are secure.
# single lower case letters are type 0 variables, and the 0
# is not displayed.  Multiletter type 0 variables are as before.
# declareproperty and declarerelation commands added
# in addition, declarefunction and declareoperation
# installed declaration and definition of operators with strongly
# cantorian range, a serious strengthening of the logic which requires
# testing.

# the s.c. upgrade is not perhaps of pedagogical value, but it's
# very useful.  It may make the Frege implementation practical.

# changed instantiables from x_3! to x$3.  $ is reserved only for this purpose.
# this is pedagogical.

# I *appear* to have fixed the index stuttering problem -- by having nextobjectindex
# updated only by the parser and the display functions.

# I need to ensure that propositional variables, instantiables, and indexed
# operators cannot appear in definitions.  This is implemented:  propositional
# variables and all indexed identifiers (including indexed constants) are excluded
# from definitions and axioms.  Theorems with all these items in them can be
# proved, but they can only be proved using full generality.

# fiddled with presentation of proofs in Showall and got better but not perfect order.

# Major wish list items:

# ability to save log files needed for pedagogy.

# autoprune -- include genealogy of formulas in sequents, and drop ones that arent used.

# powerful matching -- provide option of carrying out substitutions of instantiables which will make
# a Done command or an equality matching command succeed.

# alpha equivalence.

# comments are needed for higher order logic rules.

# set abstracts with non variables on the left.   These have real teaching value.

# substitutions using biconditionals.  These have real teaching value.

# Major testing is needed of the new s.c. type machinery and of the tighter object indexing.

# character classes of significance

def isnumeral(c):  return '0'==c or '0'<c<'9' or '9'==c

def islower(c):  return 'a' == c or 'a'<c<'z' or 'z'== c

def isupper(c):  return 'A'==c or 'A'<c<'Z' or 'Z'==c

def isspecial(c):  return  c in {'~','#','%','^','&','*','-','+','=',',',';',':','<','>','/','`','@',':','|','`'}

# returns the initial segment of the second string argument
#  which passes the character class test represented by the first argument,
#  tupled with its length.

def getinit(f,x):
```

```
    if len(x)==0:  return ""
    y=""
    for n in range(len(x)):
        if not f(x[n]): return y
        y=y+x[n]
    return y

# tests for characters which can occur in Python Marcel text

def isgoodchar(c):  return islower(c) or isupper(c) or isspecial(c) or isnumeral(c) or c in {'$','(',')','{','}','[',']','_','!','?','.'}

# strips all non-Python Marcel characters (such as whitespace)
# out of a string

def strip(s):
    if len(s)==0:  return s
    if len(s)==1 and isgoodchar(s[0]): return s
    if len(s)==1: return ''
    if (isupper(s[0]) or isspecial(s[0])) and not isgoodchar(s[1]):
        return s[0]+'.'+strip(s[2:])
    if isgoodchar(s[0]): return s[0]+strip(s[1:])
    if not isgoodchar(s[0]): return strip(s[1:])

# returns the first Python Marcel identifier in a string
# tupled with its length

# strings of lower case not followed by ? ! _ or numeral are constants
# strings of lower case followed by a numeral are variables;
#  the numeral is its type
# strings of lower case followed by ? are propositional constants
# strings of lower case followed by an underscore and numeral and not !
# are constants
# any constant form followed by ! is an instantiable (an unknown that the user
# can set)
# strings of upper case or strings of special characters, possibly followed
# by an underscore and numeral are operators.

nextobjectindex=1

def getident(x):
    global nextobjectindex
    if x=="":  return ([],0)
    if islower(x[0]):
        y=getinit(islower,x)
        if len(y)==len(x):
            if len(y)==1: return[['variable',y,'0'],1]
            return(['constant',y],len(y))
        if isnumeral(x[len(y)]):
            z=getinit(isnumeral,x[len(y):])
            return (['variable',y,z],len(y)+len(z))
        if x[len(y)]=='_' and len(x)>len(y)+1 and isnumeral(x[len(y)+1]):
            z=getinit(isnumeral,x[len(y)+1:])
            if int(z)+1>nextobjectindex: nextobjectindex=int(z)+1
            #if (not len(x)==len(y)+len(z)+1) and x[len(y)+len(z)+1]=='!':
                #return(['instantiable',y,z],len(y)+len(z)+2)
            return(['constant',y,z],len(y)+len(z)+1)
        if x[len(y)]=='$' and len(x)>len(y)+1 and isnumeral(x[len(y)+1]):
            z=getinit(isnumeral,x[len(y)+1:])
            if int(z)+1>nextobjectindex: nextobjectindex=int(z)+1
            #if (not len(x)==len(y)+len(z)+1) and x[len(y)+len(z)+1]=='!':
                #return(['instantiable',y,z],len(y)+len(z)+2)
            return(['instantiable',y,z],len(y)+len(z)+1)
        if x[len(y)]=='?':  return(['propvar',y],len(y)+1)
        if x[len(y)]=='$':  return(['instantiable',y],len(y)+1)
        if len(y)==1:  return (['variable',y,'0'],1)
        return(['constant',y],len(y))
    if isupper(x[0]):
        y=getinit(isupper,x)
        if len(y)==len(x):  return(['operator',y],len(y))
        if x[len(y)]=='.':  return(['operator',y],len(y)+1)
        if x[len(y)]=='_' and len(x)>len(y)+1 and isnumeral(x[len(y)+1]):
            z=getinit(isnumeral,x[len(y)+1:])
            return(['operator',y,z],len(y)+len(z)+1)
        return(['operator',y],len(y))
    if isspecial(x[0]):
        y=getinit(isspecial,x)
        if len(y)==len(x):  return(['operator',y],len(y))
```

```
        if x[len(y)]=='.':  return(['operator',y],len(y)+1)
        if x[len(y)]=='_' and len(x)>len(y)+1 and isnumeral(x[len(y)+1]):
            z=getinit(isnumeral,x[len(y)+1:])
            return(['operator',y,z],len(y)+len(z)+1)
        return(['operator',y],len(y))
    if isnumeral(x[0]):
        z=getinit(isnumeral,x)
        return (['constant',z],len(z))
    return ([],0)

# tests for an identifier; I have not used this.

def isident(x):
    if len(x)==0: return False
    if x[0]=='constant':
        if len(x)==2: return getinit(islower,x[1])==x[1]
        if len(x)==3:  return getinit(islower,x[1])==x[1] and getinit(isnumeral,x[2])==x[2]
        return False
    if x[0]=='operator':
        if len(x)==2:  return getinit(isupper,x[1])==x[1] or getinit(isspecial,x[1])==x[1]
        if len(x)==3:  return (getinit(isupper,x[1])==x[1] or getinit(isspecial,x[1])==x[1]) and     getinit(isnumeral,x[2])==x[2]
        return False
    if x[0]=='instantiable' or x[0]=='propvar': return len(x) == 2 and getinit(islower,x[1])==x[1]
    return False

#parentheses braces and brackets dictionary, closing forms indexed by their opening forms.
#the various paired forms do not have differing roles in input in Python Marcel, though
# they may in output

pairedforms ={'(':')','[':']','{':'}'}

# parse a non infix term

def parseterm1(x):
    global nextobjectindex
    if x[0] in pairedforms.keys():
        if len(x)==1:  return ([],0)
        z=parseterm(x[1:])
        if z[0]==[]:  return ([],0)
        t=z[0]
        n=z[1]
        if not x[n+1]==pairedforms[x[0]]:  return ([],0)
        return (['parenthesis',t],n+2)
    z=getident(x)
    if z[0]==[]: return ([],0)
    t=z[0]
    n=z[1]
    if not t[0]=='operator':  return z
    w=parseterm(x[n:])
    if w[0]==[]:  return ([],0)
    return (['unary',t,w[0]],n+w[1])

# parse an infix term using APL precedence convention (
# everything has same precedence and groups to the right)

def parseterm(x):
    global nextobjectindex
    z=parseterm1(x)
    if z[0]==[]:  return([],0)
    t=z[0]
    n=z[1]
    if len(x)==n:return z
    if not(isspecial(x[n]) or isupper(x[n])):  return z
    w=getident(x[n:])
    o=w[0]
    m=w[1]
    u=parseterm(x[n+m:])
    if u[0]==[]:  return([],0)
    s=u[0]
    p=u[1]
    return (['binary',o,t,s],n+m+p)

# parse a stripped term (the earlier functions assume input is stripped)

def parse1(x):  return parseterm(strip(x))[0]

# a fake value for the list of precedences
```

```
precedences = {'~':2,'&':0,'->':0,'V':0,'==':0,'=/=':0,':':0,':>':0,'`':0,'.':0,'E':0,'=':0}

# compute the precedence of an operator -- default is 0.

def prec(o):
    try: return precedences[o]
    except KeyError: return 0

#lowest propositional connective precedence

floor1=2

#lowest relation precedence

floor2=4

#lowest operation precedence

floor3=6

#set precedence of an operator to a fixed nonnegative value.
#precedence is determined by the text part of an operator (not
# any numerical index) and if unary and binary operators have same
# text they have same precedence.

floor4=9

#the precedence of ` (function application)

#even precedences group to right and odd to left

def setprec(o,n):
     if not n<0: precedences[o]=n

#set precedence of first operator to be same as that of second

def setprecsame(a,b):  setprec(a,prec(b))

#raise all precedences above a certain value by two to allow insertion

def raiseprec1(a,n):
    global precedences
    if prec(a)>n:  setprec(a,prec(a)+2)

#set precedence of first operator to be just above or just below
# that of second operator and of given parity

# probably these commands should check the types of
# the operators whose precedences are being related.

def setprecevenabove(a,b):
     global floor1
     global floor2
     global floor3
     global floor4
     n=prec(b)
     if floor1>n: floor1=floor1+2
     if floor2>n: floor2=floor2+2
     if floor3>n: floor3=floor3+2
     if floor4>n: floor4=floor4+2
     def F(c): raiseprec1(c,n)
     list(map(F,list(precedences.keys())))
     setprec(a,prec(b)+2-prec(b)%2)
def setprecoddabove(a,b):
     global floor1
     global floor2
     global floor3
     global floor4
     n=prec(b)
     if floor1>n: floor1=floor1+2
     if floor2>n: floor2=floor2+2
     if floor3>n: floor3=floor3+2
     if floor4>n: floor4=floor4+2

     def F(c): raiseprec1(c,n)
     list(map(F,list(precedences.keys())))
```

```
     setprec(a,prec(b)+1+prec(b)%2)
def setprecevenbelow(a,b):
     global floor1
     global floor2
     global floor3
     global floor4
     n=prec(b)-1
     if floor1>n: floor1=floor1+2
     if floor2>n: floor2=floor2+2
     if floor3>n: floor3=floor3+2
     if floor4>n: floor4=floor4+2

     if not n<0:
       def F(c): raiseprec1(c,n)
       list(map(F,list(precedences.keys())))
       setprec(a,prec(b)-2+prec(b)%2)
def setprecoddbelow(a,b):
     global floor1
     global floor2
     global floor3
     global floor4
     n=prec(b)-1
     if floor1>n: floor1=floor1+2
     if floor2>n: floor2=floor2+2
     if floor3>n: floor3=floor3+2
     if floor4>n: floor4=floor4+2

     if not n<0:
        def F(c): raiseprec1(c,n)
        list(map(F,list(precedences.keys())))
        setprec(a,prec(b)-1-prec(b)%2)

#  correct grouping in an APL term following precedence.  Parentheses
# left undisturbed.

def stickiness(T):
    if T[0]=='unary' or T[0]=='binary':  return prec(T[1][1])
    return(-1)

def regroup(T):
    global nextobjectindex
    if len(T)==0:  return T
    if T[0]=='unary':
        if stickiness(T[2])==-1: return T
        U=regroup(T[2])
        if not U[0]=='binary':  return ['unary',T[1],U]
        if stickiness(T) > stickiness(U) or (stickiness(T)==stickiness(U) and stickiness(T)%2==1):
            return regroup(['binary',U[1],regroup(['unary',T[1],U[2]]),U[3]])
        return ['unary',T[1],U]

    if T[0]=='binary':
        U=regroup(T[2])
        V=regroup(T[3])
        if (not stickiness(U)==-1) and (stickiness(T) > stickiness(U) or (stickiness(T)==stickiness(U) and stickiness(T)%2==0)):
            if U[0] == 'unary':  return regroup(['unary',U[1],regroup(['binary',T[1],U[2],V])])
            if U[0] == 'binary':  return regroup(['binary',U[1],U[2],regroup(['binary',T[1],U[3],V])])
        if (not stickiness(V)==-1) and (stickiness(T) > stickiness(V) or (stickiness(T)==stickiness(V) and stickiness(T)%2==1)):
            if V[0] == 'unary': return ['binary',T[1],U,V]
            if V[0] == 'binary': return regroup(['binary',V[1],regroup(['binary',T[1],U,V[2]]),V[3]])
        return(['binary',T[1],U,V])


    return T

#regroup inside parentheses then remove them.

def deparen(t):
    global nextobjectindex
    if len(t)==0:  return t
    if t[0]=='parenthesis':  return deparen(regroup (t[1]))
    if t[0]=='unary':  return ['unary',t[1],deparen(t[2])]
    if t[0]=='binary':  return ['binary',t[1],deparen(t[2]),deparen(t[3])]
    return t

# the final parser:  strip string, then parse with APL precedence, then regroup
# and deparenthesize
```

```
def parse(T):
    global nextobjectindex
    return deparen(regroup(parse1(T)))

#DISPLAY FUNCTION  -- with and without indentation and line breaks

# in output use parentheses for propositional grouping,
# brackets for object grouping, braces for abstractions?  This would
# require introduction of the operator type dictionary

# type declaration list for operators.  We assume that subscripted variants
# of operators have the same type signature.

# this list also tells us whether operators are unary or binary

optypes={':':['prop',0,0]}

def declareunary(o,t1,t2):
    if o in list(optypes.keys()): return o+' already declared.'
    if o[0]==':':  return 'bad' #reserved for abstractors
    optypes[o]=[t1,t2]
    if t1=='prop' and t2=='prop':
        setprec(o,floor1)
        return 'done'
    if t1=='prop':
        setprec(o,floor2)
        return 'done'
    setprec(o,floor3)
    return 'done'

def declareproperty(P):
    return declareunary(P,'prop',0)

def declarefunction(F):
    return declareunary(F,0,0)

def declarescfunction(F):
    return declareunary(F,'sc',0)

def declaretypedfunction(F,n):
    return declareunary(F,n+abs(n),abs(n))

def declaretypedscfunction(F,n):
    return declareunary(F,'sc',n+abs(n),abs(n))

def declarebinary(o,t1,t2,t3):
    if o in list(optypes.keys()): return o+' already declared.'
    if o[0]==':':  return 'bad' #reserved for abstractors
    optypes[o]=[t1,t2,t3]
    if t1=='prop' and t2=='prop':
        if not t3 == 'prop':  return 'bad'
        setprec(o,floor1)
        return 'done'
    if t1=='prop':
        setprec(o,floor2)
        return 'done'
    setprec(o,floor3)
    return 'done'

def declarerelation(R):
    return declarebinary(R,'prop',0,0)

def declareoperation(O):
    return declarebinary(O,0,0,0);

def declarescoperation(O):
    return declarebinary(O,'sc',0,0)

def declaretypedrelation(R,n):
    return declarebinary(R,'prop',abs(n),n+abs(n))

def declaretypedoperation(O,m,n):
    return declarebinary(O,abs(m)+abs(n)+m+n,abs(m)+abs(n)+n,abs(m)+abs(n)+m);

def declaretypedscoperation(O,n):
    return declarebinary(O,'sc',abs(n)+n,abs(n))
```

```
declaredconstants = []

def Declareconstant(s):
    global declaredconstants
    if s in list(declaredconstants): return s+' already declared.'
    if not(getident(s)[0]==['constant',s]):  return s+' cannot be a declared constant'
    declaredconstants=[s]+declaredconstants

# basic printing function


def leftunary(T):
    if len(T)==0:  return False
    return T[0]=='unary' or (T[0]=='binary' and leftunary(T[2]))

def openparen(T):
    if T[0]=='unary':
        if optypes[T[1][1]][0]=='prop':  return '('
        return '['
    if T[0]=='binary':
        if T[1][1][0]==':':  return '{'
        if optypes[T[1][1]][0]=='prop':  return '('
        return '['
    return ''

def closeparen(T):
    if T[0]=='unary':
        if optypes[T[1][1]][0]=='prop':  return ')'
        return ']'
    if T[0]=='binary':
        if T[1][1][0]==':':  return '}'
        if optypes[T[1][1]][0]=='prop':  return ')'
        return ']'
    return ''



def termprint(T):
    global nextobjectindex
    if len(T)==0: return 'Parse error!'
    if (T[0]== 'constant' or T[0]=='operator') and len(T)==2:  return T[1]
    if (T[0]=='constant' or T[0]=='operator') and len(T)==3:
        if int(str(T[2]))+1>nextobjectindex:  nextobjectindex=int(str(T[2]))+1
        return T[1]+'_'+str(T[2])
    if T[0]=='propvar':  return T[1]+'?'
    if T[0]=='instantiable' and len(T)==2:  return T[1]+'$'
    if T[0]=='instantiable' and len(T)==3:
       if int(str(T[2]))+1>nextobjectindex:  nextobjectindex=int(str(T[2]))+1
       return T[1]+'$'+str(T[2])
    if T[0]=='variable' and len(T[1])==1 and T[2]=='0':  return T[1]
    if T[0]=='variable':
       return T[1]+str(T[2])
    if T[0]=='unary':
         if T[2][0]=='unary': return termprint(T[1])+' '+termprint(T[2])  #used dot here in original version
         if stickiness(T[2])==-1:  return termprint(T[1])+termprint(T[2])
         if stickiness(T)>stickiness(T[2]) or (stickiness(T)==stickiness(T[2]) and stickiness(T[2])%2==1):
              return termprint(T[1])+openparen(T[2])+termprint(T[2])+closeparen(T[2])
         return termprint(T[1])+termprint(T[2])
    if T[0]=='binary':
        parens1=['','']
        if stickiness(T)>stickiness(T[2]) or (stickiness(T)==stickiness(T[2]) and stickiness(T)%2==0): parens1=[openparen(T[2]),closeparen(T[2])+' ']
        if stickiness(T[2])==-1: parens1=['',' ']
        parens2=[' ','']
        if stickiness(T)>stickiness(T[3]) or (stickiness(T)==stickiness(T[3]) and stickiness(T)%2==1):  parens2=[' '+openparen(T[3]),closeparen(T[3])]
        if stickiness(T[3])==-1: parens2=[' ','']
        #if leftunary(T[3]) and parens2==['','']: parens2=['.',''] this line was for inserting dots
        #for a compact version, eliminate all the spaces and put the dots back in
        return parens1[0]+termprint(T[2])+parens1[1]+termprint(T[1])+parens2[0]+termprint(T[3])+parens2[1]

termprint(parse('x+y'))

#note new use of . to break between an operator and a following unary operator when
#parentheses do not otherwise handle it.  From the parser's standpoint, a period ends
#an operator.  The printing function inserts a dot precisely when it would otherwise
#juxtapose an operator and a following unary operator. This means that . cannot appear
#in an operator.
```

```
# use of . is now entirely optional and it is never displayed.


#TYPING ALGORITHM -- need mechanisms for declaring operator types.
#should also immediately enable definition mechanism.  It would be useful
#to be able to enforce sensible precedence order for operators when they are
#declared, based on operator type.

#Substitution (and matching?)  Privileges of pairing come up here.

#Sequent manipulations and viewing.

#Theorem application (using matching).
# Idea that we can support generality over operators as well as constants.
# representation of operators as a special sort of abstract useful here.

# typing algorithm.  We first need to have a dictionary of types for
# operators.  Format is [output type, input type 1(, input type 2)]
# the possible types are 'bad', 'prop', and integers (all of which stand
# for type 'object' indifferently).



# possible type outputs for a term:  'prop', an integer, 'constant', 'bad'.

# free variable list function needed.

#decompose terms into components

def opof(T):
    if T[0]=='unary' or T[0]=='binary':  return T[1]
    return 'none'

def term1(T):
    if T[0]=='binary' or T[0]=='unary': return T[2]
    return 'none'

def term2(T):
    if T[0]=='binary':  return T[3]
    return 'none'

#this function implicitly assumes that abstractions have single variables on left

def freevars(T):
    if len(T)==0:  return []
    if T[0] == 'variable':  return [T]
    if T[0] == 'unary':  return freevars(T[2])
    if T[0] == 'binary':
        if opof(T)[1] == ':' or opof(T)[1] == ':>':
            def f(x):  return not x==T[2]
            return list(filter(f,freevars(T[3])))
        return freevars(T[2])+freevars(T[3])
    return []

#the function freevars2 ignores free variables directly tagged with
# s.c. output unary operators.  It is used to recognize sets
# restricted to s.c. types.

def scvariable(T):
    if not T[0] == 'unary': return False
    if not T[2][0] == 'variable': return False
    op=opof(T)
    if not op[1] in list(optypes.keys()):  return False
    opt = optypes[op[1]]
    if not len(opt) == 2:  return False
    if not opt[0] == 'sc':  return False
    return True


def freevars2(T):
    if len(T)==0:  return []
    if T[0] == 'variable':  return [T]
    if scvariable(T):  return []
    if T[0] == 'unary':  return freevars(T[2])
    if T[0] == 'binary':
        if opof(T)[1] == ':' or opof(T)[1] == ':>':
            def f(x):  return not x==T[2]
```

```
            return list(filter(f,freevars2(T[3])))
        return freevars2(T[2])+freevars2(T[3])
    return []

# the type algorithm.  It also enforces unary/binary operator distinctions
# and syntax of abstraction terms.

# added option of unary and binary functions with strongly cantorian range
# (output type 'sc')

#it is important to notice that 'constant' means shiftable in type,
# i.e, bounded in some s.c. range,
# not literally constant, though that is included.

def type(T):
    if len(T)==0:  return 'bad'
    if T[0]=='constant':
        if len(T)==3:  return 'constant'
        if len(T)==2 and T[1] in declaredconstants: return 'constant'
        return 'bad'
    if T[0] == 'variable':  return int(T[2])
    if T[0] == 'propvar':  return 'prop'
    if T[0] == 'unary':
        tt=type(T[2])
        if tt == 'bad':  return 'bad'
        op=opof(T)
        if not op[1] in list(optypes.keys()):  return 'bad'
        opt=optypes[op[1]]
        if not len(opt)==2: return 'bad'
        if opt[1]=='prop':
            if tt=='prop':  return opt[0]
            return 'bad'
        #we assume at this point that input type is integer
        if opt[0]=='prop':  return 'prop'
        if opt[0]=='sc':  return 'constant'
        if tt=='constant':  return 'constant'
        return tt + (opt[0]-opt[1])
    if T[0] == 'binary':
        tt1=type(T[2])
        #print ('tt1 is'+str(tt1))
        tt2=type(T[3])
        #print ('tt2 is'+str(tt2))
        if tt1=='bad' or tt2=='bad':  return 'bad'
        op=opof(T)
        #print('operation is'+str(op))
        if op[1]==':':
            if not tt2=='prop':  return 'bad'
            if not T[2][0]=='variable':  return 'bad'
            if freevars(T)==[]:  return 'constant'
            if not T[2] in freevars2(T[3]): return 'constant'
            return type(T[2])+1
        #ability to recognize that sets are of an s.c. type is needed
        if op[1]==':>':
            if tt2=='prop':  return 'bad'
            if not T[2][0]=='variable':  return 'bad'
            if freevars(T)==[]:  return 'constant'
            if not T[2] in freevars2(T[3]) and tt2=='constant':  return 'constant'
            if not tt2==type(T[2]):  return 'bad'
            return tt2+1
        if not op[1] in list(optypes.keys()):  return 'bad'
        opt=optypes[op[1]]
        #print('operation type is '+str(opt))
        if not len(opt)==3:  return 'bad'
        if opt[1]=='prop' and not tt1=='prop':  return 'bad'
        if opt[2]=='prop' and not tt2=='prop':  return 'bad'
        if opt[0]=='prop':
            if opt[1]=='prop' or opt[2]=='prop':  return 'prop'
            if tt1=='constant' or tt2=='constant':  return 'prop'
            if tt1-tt2==opt[1]-opt[2]:  return 'prop'
            return 'bad'
        if opt[0]=='sc':
            #if opt[1]=='prop' or opt[2]=='prop':  return 'prop'
            if tt1=='constant' or tt2=='constant':  return 'constant'
            if tt1-tt2==opt[1]-opt[2]:  return 'constant'
            return 'bad'
        if opt[1]=='prop' or tt1=='constant':
            if opt[2]=='prop' or tt2=='constant': return 'constant'
```

```
            return tt2+(opt[0]-opt[2])
        return tt1+(opt[0]-opt[1])
    #for any other atomic term...
    return 'constant'

# I should set up the declaration commands so that precedences are controlled,
# propositional connectives lowest, then relations, then functions.  An idea:
# set threshhold values, raised appropriately when precedences are inserted,
# and automatically set by the declaration commands, then do not allow precedences
# to be lowered.

declareunary('A','prop',0)
setprec('A',0)
declareunary('E','prop',0)
setprec('E',0)
declarebinary('=','prop',0,0)
declarebinary('IN','prop',0,1)
declareunary('~','prop','prop')
declareunary('* ~','prop','prop')
declarebinary('V','prop','prop','prop')
declarebinary('->','prop','prop','prop')
declarebinary('==','prop','prop','prop')
declarebinary('&','prop','prop','prop')
declarebinary('`',0,1,0)
declareunary('THE',0,1)
setprec('THE',0)

setprec('`',floor4)
setprecevenabove('->','==')
setprecoddabove('V','->')
setprecoddabove('&','V')
setprecoddabove('~','&')
setprecsame('* ~','~')

# substitution is next

def substitute(t,v,T):
     global theproof
     if not (v[0]=='variable' or v[0]=='constant' or v[0]=='instantiable'):  return T
     if not type(t)=='constant':  return T
     if type(T)=='bad': return T
     if termprint(T)==termprint(v): return t
     if T[0]=='unary':  return ['unary',T[1],substitute(t,v,T[2])]
     if T[0]=='binary':
         if T[1][1]==':' or T[1][1]==':>':
             if T[2]==v:  return T
             return ['binary',T[1],substitute(t,v,T[2]),substitute(t,v,T[3])]
         return ['binary',T[1],substitute(t,v,T[2]),substitute(t,v,T[3])]
     return T

# rules for use in equality substitutions
# versions usable for propositions with biconditionals are very similar

# equality rules on left will also require rotations of all assumptions except the first,
# to bring the desired equation into second position

# substitute one constant term for another everywhere

def strongsubstitute(a,b,T):
     if not type((a))=='constant' or not type((b))=='constant':  return T
     if type(T)=='bad': return T
     if termprint(T)==termprint(a):  return b
     if T[0]=='unary':  return ['unary',T[1],strongsubstitute(a,b,T[2])]
     if T[0]=='binary':  return ['binary',T[1],strongsubstitute(a,b,T[2]),strongsubstitute(a,b,T[3])]
     return T

# substitute leftmost occurrence only

def strongsubstitute1(a,b,T):
     if not type((a))=='constant' or not type((b))=='constant':  return T
     if type(T)=='bad': return T
     if termprint(T)==termprint(a):  return b
     if termprint(T)==termprint(b):  return b
     if T[0]=='unary':  return ['unary',T[1],strongsubstitute1(a,b,T[2])]
     if T[0]=='binary':
         A=strongsubstitute1(a,b,T[2])
         if termprint(A)==termprint(T[2]):  return ['binary',T[1],T[2],strongsubstitute1(a,b,T[3])]
```

```
        return ['binary',T[1],A,T[3]]
    return T

# substitute rightmost occurrence only

def strongsubstitute2(a,b,T):
    if not type((a))=='constant' or not type((b))=='constant':  return T
    if type(T)=='bad': return T
    if termprint((T))==termprint((a)):  return b
    if termprint(T)==termprint(b):  return b
    if T[0]=='unary':  return ['unary',T[1],strongsubstitute(a,b,T[2])]
    if T[0]=='binary':
        A=strongsubstitute2(a,b,T[3])
        if termprint(A)==termprint(T[3]):  return ['binary',T[1],strongsubstitute2(a,b,T[2]),T[3]]
        return ['binary',T[1],T[2],A]
    return T

oldobjectindex=0
thedifferential=0

def instsubstitute0(T):
    global oldobjectindex
    if T[0]=='unary':  return ['unary',T[1],instsubstitute0(T[2])]
    if T[0]=='binary':  return ['binary',T[1],instsubstitute0(T[2]),instsubstitute0(T[3])]
    if (T[0]=='instantiable') and len(T)==3:
        return['instantiable',T[1],str(int(T[2])+oldobjectindex)]
    return T

def propsubstitute(t,v,T):
    global theproof
    if not (v[0]=='propvar'):  return T
    if not type(t)=='prop':  return T
    if not freevars(t)==[]:  return T
    if type(T)=='bad': return T
    if termprint(T)==termprint(v): return t
    if T[0]=='unary':  return ['unary',T[1],propsubstitute(t,v,T[2])]
    if T[0]=='binary':
        if T[1][1]==':' or T[1][1]=='>':
            if termprint(T[2])==termprint(v):  return T
            return ['binary',T[1],propsubstitute(t,v,T[2]),propsubstitute(t,v,T[3])]
        return ['binary',T[1],propsubstitute(t,v,T[2]),propsubstitute(t,v,T[3])]
    return T

def opprint(op):
    if not len(op)==2 and not len(op)==3:  return '_bad_'
    if len(op)==2:  return op[1]
    if len(op)==3:  return op[1]+'_'+op[2]

# replace an indexed operator with another operator of the same type

def opsubstitute(t,v,T):
    global theproof
    if len(v) == 0 or not (v[0]=='operator'):  return T
    if not len(v)==3:  return T
    if len(t)==0 or not (t[0]=='operator'):  return T
    if not optypes[t[1]]==optypes[v[1]]:  return T
    if T[0]=='unary':
        if opprint(T[1])==opprint(v):  return ['unary',t,opsubstitute(t,v,T[2])]
        return ['unary',T[1],opsubstitute(t,v,T[2])]
    if T[0]=='binary':
        if opprint(T[1])==opprint(v):  return ['binary',t,opsubstitute(t,v,T[2]),opsubstitute(t,v,T[3])]
        return ['binary',T[1],opsubstitute(t,v,T[2]),opsubstitute(t,v,T[3])]
    return T

def testsub(t,v,T):  return termprint(opsubstitute(getident(t)[0],getident(v)[0],parse(T)))

#  there should also be a substitution function for propositions.
#  Propositions substituted for propositional variables need to contain
#  no free object variables, for the same reason that objects substituted
#  need to be of type constant.  Otherwise substitutions could break
#  stratification.

# basic functions for sequent display

def printproplist0(L,n):
    if len(L)==0: return '\n'
    if not type(L[0])=='prop':  return '\n'
```

```
    return str(n)+":  "+termprint(L[0])+'\n'+printproplist0(L[1:],n+1)

def printproplist(L):  return printproplist0(L,1)

def printproplistar0(L,n):
    if len(L)==0: return '\n'
    if not type(L[0])=='prop':  return '\n'
    return str(n)+"*:  "+termprint(L[0])+'\n'+printproplistar0(L[1:],n+1)

def printproplistar(L):  return printproplistar0(L,1)

# a sequent is a natural number identifier and two lists of propositions.

oneconclusion=False

oneconclusion2 = False

def Oneconclusion():
    global oneconclusion
    oneconclusion=True

def Manyconclusions():
    global oneconclusion
    oneconclusion=False

def Oneconclusion2():
    global oneconclusion2
    oneconclusion2=True

def Manyconclusions2():
    global oneconclusion
    oneconclusion2=False

def pseudonegate(T):  return ['unary',['operator','~'],T]

def pseudolist(L):
    if len(L)==0:  return L
    return [pseudonegate(L[0])]+pseudolist(L[1:])

def printsequent(S):

    if oneconclusion and not len(S[2])==0:
        return str(S[0])+':\n\n'+printproplist(S[1])+printproplistar0(pseudolist(S[2][1:]),2)+"------------------\n\n"+printproplist([S[2][0]])+'\n\n'
    if oneconclusion and len(S[2])==0:
        return str(S[0])+':\n\n'+printproplist(S[1])+"------------------\n\n_|_\n\n"
    return str(S[0])+':\n\n'+printproplist(S[1])+"------------------\n\n"+printproplist(S[2])+'\n\n'

def negate(T):
    if len(T)<2:  return ['unary',['operator','~'],T]
    if T[0]=='unary' and T[1]==['operator','~']:  return T[2]
    return ['unary',['operator','~'],T]

def neglist(L):
    if len(L)==0:  return L
    return [negate(L[0])]+neglist(L[1:])

def transformsequent(S):
    if not oneconclusion2:  return S
    if len(S[2])==0:  return S
    return [S[0],S[1]+neglist(S[2][1:]),[S[2][0]]]

# a proof is a sequent paired with 'goal' or a list of proofs to be completed.
# a proof is complete if none of its subproofs contain 'goal'.

# we will want genealogy data in sequents or proofs somewhere in order to support
# automatic pruning.  Make simple version now:  but post-engineering is to be expected.

nextsequent=0

def makesequent(L,M,comment):
    global nextsequent
    nextsequent=nextsequent+1
    return (transformsequent(['Line '+str(nextsequent)+comment,L,M]))

theproof=[[0,[],[]],'goal']

def Start(p):
```

```
    global nextobjectindex
    nextobjectindex=1
    if not freevars(parse(p))==[]:
        print('bad')
        return 'bad'
    if not type(parse(p))=='prop':
        print('bad')
        return 'bad'
    global nextsequent
    nextsequent=0
    global theproof
    theproof=[makesequent([],[parse(p)],''),'goal']
    print(printsequent(theproof[0]))


#is a given proof structure completely resolved?

def completeproof(P):
    global theproof
    if P[1]=='goal':  return False
    for Q in P[1]:
        if not completeproof(Q): return False
    return True

# move the first available goal into leftmost position

def onerotate(P):
    if P[1]=='goal':  return P
    if P[1]==[]:  return P
    if len(P[1])==1:  return P
    return ([P[0],P[1][1:]+[P[1][0]]])

def rotateproof(P):
    global theproof
    if completeproof(P): return onerotate(P)
    if P[1]=='goal':  return P
    if completeproof(P[1][0]): return rotateproof ([P[0],P[1][1:]+[onerotate(P[1][0])]])
    return [P[0],[rotateproof(P[1][0])]+P[1][1:]]

# a not very efficient way to traverse the proof tree

def rotateproof2(P):
    if completeproof(P):  return P
    if P[1]=='goal':  return P
    return(rotateproof3(rotateproof([P[0],P[1][1:]+[P[1][0]]])))

def rotateproof3(P):
     return[P[0],[rotateproof2(P[1][0])]+P[1][1:]]

# show the leftmost goal

def shownextgoal(P):
    global theproof
    if P[1]=='goal':  print(printsequent(P[0]))
    if P[1]==[]:  print('Q. E. D.')
    if not(P[1]=='goal') and not(len(P[1])==0):  shownextgoal(P[1][0])

def Nextgoal():
    global theproof
    theproof=rotateproof2(theproof)
    shownextgoal(theproof)

# this is a temp definition, I need at least equality up to alpha conversion.  Or do I?
#  equality of alpha equivalent abstractions is easy to prove, and a given constant abstraction
# is most likely to occur via a definition!

def matches(P,Q):  return termprint(P)==termprint(Q)

# apply a function to the leftmost goal

# the function f takes the sequent in the leftmost goal to a list of proofs

def leftapply(f,P):
    global theproof
    if P[1]==[]: return P
    if P[1]=='goal':  return rotateproof([P[0],f(P[0])])
    return rotateproof([P[0],[leftapply(f,P[1][0])]+P[1][1:]])
```

```
def act(f):
    global theproof
    theproof=leftapply(f,theproof)
    shownextgoal(theproof)

# the function f takes the sequent in the leftmost goal to a single proof

def leftapply2(f,P):
    global theproof
    if P[1]==[]: return P
    if P[1]=='goal':  return f(P[0])
    return rotateproof([P[0],[leftapply2(f,P[1][0])]+P[1][1:]])

def act2(f):
    global theproof
    theproof=leftapply2(f,theproof)
    shownextgoal(theproof)


#  we need a done function which takes a sequent with first
# props on both sides matching to the empty list of proofs.

def done(S):
    if len(S[1])==0 or len(S[2])==0:  return 'goal'
    if matches(S[1][0],S[2][0]):  return []
    return 'goal'

def Look():
    global theproof
    shownextgoal(theproof)

def Done():  act(done)

def distribute(L,S):
    if len(L)==0:  return []
    return [[makesequent(L[0][0]+S[1],L[0][1]+S[2],L[0][2]),'goal']]+distribute(L[1:],S)

def leftaction(S):
    if len(S[1])== 0:
        print('left rule fails to apply\n')
        return 'goal'
    if leftexpand(S[1][0])=='fail':
        print('left rule fails to apply\n')
        return 'goal'
    return distribute(leftexpand(S[1][0]),[S[0],S[1][1:],S[2]])

def rightaction(S):
    if len(S[2])== 0:
        print('right rule fails to apply\n')
        return 'goal'
    if rightexpand(S[2][0])=='fail':
        print('right rule fails to apply\n')
        return 'goal'
    return distribute(rightexpand(S[2][0]),[S[0],S[1],S[2][1:]])

def Left():  act(leftaction)

def Right():  act(rightaction)

def leftprune(S):  return [[S[0],S[1][1:],S[2]],'goal']

def rightprune(S):  return [[S[0],S[1],S[2][1:]],'goal']

def Pruneleft():  act2(leftprune)

def Pruneright():  act2(rightprune)

# definition facility

Definitions = {}

# the argument of the define command is a list, name of defined operator
# followed by one or two variables followed by body
# installed ability to define operators with strongly cantorian range.

# we cannot make a definition involving propositional variables, instantiables, or indexed operators.
```

```
# this also excludes indexed constants, though strictly speaking they could be permitted.

def defsafe(T):

    if len(T)==0: return False
    if (T[0]== 'constant' or T[0]=='operator') and len(T)==2:  return True
    if (T[0]=='constant' or T[0]=='operator') and len(T)==3:  return False
    if T[0]=='propvar':  return False
    if T[0]=='instantiable' and len(T)==2:  return False
    if T[0]=='instantiable' and len(T)==3:  return False

    if T[0]=='variable' and len(T[1])==1 and T[2]=='0':  return True
    if T[0]=='variable':
       return True
    if T[0]=='unary':  return defsafe(T[2]) and len(T[1])==2

    if T[0]=='binary':
        return defsafe(T[2]) and defsafe(T[3]) and len(T[1])==2
    return False

def Define(L):
        ident=getident(L[0])[0]
        if (not len(ident)==2) or (not ident[0]=='operator'):  return 'bad'
        if ident[1] in list(optypes.keys()):  return 'bad'
        if len(ident)==3:  return 'bad'
        if len(L)==3:
            var1 = getident(L[1])[0]
            if len(var1)==0 or not var1[0]=='variable':  return 'bad'
            body=parse(L[2])
            if type(body)=='bad':  return 'bad'
            thetype=type(body)
            if not defsafe(body):  return 'bad'
            if thetype=='constant':  thetype='sc'
            body2=substitute(['constant','.arg','1'],var1,body)
            if not freevars(body2)==[]:  return 'bad'
            Definitions[ident[1]]=body2
            declareunary(ident[1],thetype,type(var1))
            return 'done'
        if len(L)==4:
            var1=getident(L[1])[0]
            var2=getident(L[2])[0]
            if len(var1)==0 or not var1[0]=='variable':  return 'bad'
            if len(var2)==0 or not var2[0]=='variable':  return 'bad'
            body=parse(L[3])
            if type(body)=='bad':  return 'bad'
            if not defsafe(body):  return 'bad'
            thetype=type(body)
            if thetype=='constant':  thetype='sc'
            body2=substitute(['constant','.arg','1'],var1,substitute(['constant','.arg','2'],var2,body))
            Definitions[ident[1]]=body2
            declarebinary(ident[1],thetype,type(var1),type(var2))
            return 'done'
        return 'bad'

def Defineconstant(c,t):
    global declaredconstants
    ident = parse(c)
    if (not len(ident)==2) or (not ident[0]=='constant'):  return 'error'
    if ident[1] in declaredconstants:  return 'error'
    body = parse(t)
    if not(defsafe(body)):  return 'error'
    if not type(body)=='constant' or not freevars(body) == []:  return 'error'
    Declareconstant(ident[1])
    Definitions[ident[1]]=body


def defexpand(T):
    if T[0]=='constant' and len(T)==2 and T[1] in list(Definitions.keys()):
        return Definitions[T[1]]
    if T[0]=='unary':
        if T[1][1] in list(Definitions.keys()) and len(T[1])==2:
            return substitute(T[2],['constant','.arg','1'],Definitions[T[1][1]])
        return [T[0],T[1],defexpand(T[2])]
    if T[0]=='binary':
        if T[1]==['operator',''] and T[2][0]=='binary' and T[2][1]==['operator',':>']:
            return substitute(T[3],T[2][2],T[2][3])
        if T[1][1] in list(Definitions.keys()) and len(T[1])==2:
```

```
            return substitute(T[2],['constant','.arg','1'],substitute(T[3],['constant','.arg','2'],Definitions[T[1][1]]))
        return [T[0],T[1],defexpand(T[2]),defexpand(T[3])]
    return T

#functions leftaction and rightaction will contain the meat of the logic

def leftexpand(P):
    global nextobjectindex
    if P[0] == 'unary' and P[1]==['operator','~']:
        return [[[],[P[2]],'\nuse \n'+(termprint (P))+'\nby denying conclusion and proving\n'+(termprint(P[2]))]]
    if P[0] == 'binary' and P[1]==['operator','&']:
        return [[[P[2],P[3]],[],'\nuse \n'+(termprint(P))+'\nby breaking it into its parts\n'+(termprint(P[2]))+'\nand\n'+(termprint(P[3]))]]
    if P[0]=='binary' and P[1]==['operator','V']:
        return [[[P[2]],[],'\nusing hypothesis\n'+(termprint(P))+'\nfirst part: assume case 1,\n'+(termprint(P[2]))],[[P[3]],[],'\nusing hypothesis\n'+(t
    if P[0]=='binary' and P[1]==['operator','->']:
        return [[[],[P[2]],'\nuse\n'+(termprint(P))+'\n, first part, showing that\n'+(termprint(P[2]))+'\nor the desired conclusion holds'],[[P[3]],[],'\
    if P[0]=='binary' and P[1]==['operator','==']:
        firstimp=['binary',['operator','->'],P[2],P[3]]
        secondimp=['binary',['operator','->'],P[3],P[2]]
        return [[[firstimp,secondimp],[],'\nbreak biconditional assumption\n'+(termprint(P))+'\ninto its constituent implications']]
    if P[0]=='unary' and P[1]==['operator','A'] and P[2][0] == 'binary' and P[2][1]==['operator',':']:
 #       nextobjectindex=nextobjectindex+1
        return [[[substitute(['instantiable',P[2][2][1],str(nextobjectindex)],P[2][2],P[2][3]),P],[],'\nuse the universal hypothesis\n'+(termprint(P))+'
    if P[0]=='unary' and P[1]==['operator','E'] and P[2][0] == 'binary' and P[2][1]==['operator',':']:
 #       nextobjectindex=nextobjectindex+1
        return [[[substitute(['constant',P[2][2][1],str(nextobjectindex)],P[2][2],P[2][3])],[],'\nuse the existential hypothesi\n'+(termprint(P))+'\nby i
    if P[0]=='binary' and P[1]==['operator','IN']:
        if P[3][0]=='binary' and P[3][1]==['operator',':']:
            return [[[substitute(P[2],P[3][2],P[3][3])],[],'']]
    if P[0]=='binary' and P[1]==['operator','=']:
        if termprint(P[2])==termprint(P[3]):  return [[[],[]]]
        return [[[substitute(P[2],parse('x0'),substitute(P[3],parse('y0'),parse('Az1:x0INz1->y0INz1'))),P],[],'']]
    D=defexpand(P)
    if not P==D:  return [[D],[],'']]
    if (P[0]=='unary' and P[2][0]== 'unary' and P[2][1]==['operator','THE']):
         C1=substitute(P[2][2],parse('x1'),parse('Ey0:Ax0:x0INx1==x0=y0'))
         C2=substitute(P[2][2],parse('x1'),parse('Ax0:x0INx1==x0=THEx1'))
         return [[[P],[C1],''],[[C2,P],[],'']]
    if (P[0]=='binary' and P[3][0]== 'unary' and P[3][1]==['operator','THE']):
         C1=substitute(P[3][2],parse('x1'),parse('Ey0:Ax0:x0INx1==x0=y0'))
         C2=substitute(P[3][2],parse('x1'),parse('Ax0:x0INx1==x0=THEx1'))
         return [[[P],[C1],''],[[C2,P],[],'']]
    if (P[0]=='binary' and P[2][0]== 'unary' and P[2][1]==['operator','THE']):
         C1=substitute(P[2][2],parse('x1'),parse('Ey0:Ax0:x0INx1==x0=y0'))
         C2=substitute(P[2][2],parse('x1'),parse('Ax0:x0INx1==x0=THEx1'))
         return [[[P],[C1],''],[[C2,P],[],'']]
    return 'fail'

def rightexpand(P):
    global nextobjectindex
    if P[0]=='unary' and P[1]==['operator','~']:
        return [[[P[2]],[],'\nprove\n'+(termprint(P))+'\nby assuming\n'+(termprint(P[2]))+'\nand deducing a contradiction or alternative conclusion']]
    if P[0]=='binary' and P[1]==['operator','&']:
        return [[[],[P[2]],'\nprove\n'+(termprint(P))+'\nfirst part:  prove\n'+(termprint(P[2]))],[[],[P[3]],'\nprove\n'+(termprint(P))+'\nsecond part:
    if P[0]=='binary' and P[1]==['operator','V']:
        return [[[],[P[2],P[3]],'\nprove\n'+(termprint(P))+'\nby denying\n'+(termprint(P[3]))+'\nand showing\n'+(termprint(P[2]))]]
    if P[0]=='unary' and P[1]==['operator','A'] and P[2][0] == 'binary' and P[2][1]==['operator',':']:
        n = nextobjectindex
 #       nextobjectindex=nextobjectindex+1
        return[[[],[substitute(['constant',P[2][2][1],str(n)],P[2][2],P[2][3])],'\nprove the universal\n'+(termprint(P))+'\nby proving an arbitrary insta
    if P[0]=='unary' and P[1]==['operator','E'] and P[2][0] == 'binary' and P[2][1]==['operator',':']:
        n=nextobjectindex
 #       nextobjectindex=nextobjectindex+1
        return[[[],[substitute(['instantiable',P[2][2][1],str(n)],P[2][2],P[2][3]),P],'\nprove the existential \n'+(termprint(P))+'\nby introducing an in
    if P[0]=='binary' and P[1]==['operator','->']:
        return [[[P[2]],[P[3]],'\nprove\n'+(termprint(P))+'\nby assuming\n'+(termprint(P[2]))+'\nand deducing\n'+(termprint(P[3]))]]
    if P[0]=='binary' and P[1]==['operator','==']:
        return [[[P[2]],[P[3]],'\nproving biconditional\n'+(termprint(P))+'\nPart I =>\n'],[[P[3]],[P[2]],'\nproving biconditional\n'+(termprint(P))+'\nP
    if P[0]=='binary' and P[1]==['operator','IN']:
        if P[3][0]=='binary' and P[3][1]==['operator',':']:
            return [[[],[substitute(P[2],P[3][2],P[3][3])],'']]
    if P[0]=='binary' and P[1]==['operator','=']:
        if termprint(P[2])==termprint(P[3]): return []
        if P[2][0]=='binary' and P[2][1]==['operator',':'] and P[3][0]=='binary' and P[3][1]==['operator',':']:
 #           nextobjectindex=nextobjectindex+1
            A=substitute(['constant',P[2][2][1],str(nextobjectindex)],P[2][2],P[2][3])
            B=substitute(['constant',P[2][2][1],str(nextobjectindex)],P[3][2],P[3][3])
```

```
           return [[[],[[’binary’,[’operator’,’==’],A,B]],’’]]
        if P[2][0]==’binary’ and P[2][1]==[’operator’,’:>’] and P[3][0]==’binary’ and P[3][1]==[’operator’,’:>’]:
#           nextobjectindex=nextobjectindex+1
           A=substitute([’constant’,P[2][2][1],str(nextobjectindex)],P[2][2],P[2][3])
           B=substitute([’constant’,P[2][2][1],str(nextobjectindex)],P[3][2],P[3][3])
           return [[[],[[’binary’,[’operator’,’=’],A,B],],’’]]
        D=defexpand(P)
        if not P==D:  return [[[],[D],’’]]
        return [[[],[substitute(P[2],parse(’x0’),substitute(P[3],parse(’y0’),parse(’Az1:x0INz1->y0INz1’))),
               substitute((P[2]),parse(’x1’),substitute((P[3]),parse(’y1’),parse(’x1={x0:x0INx1}&y1={x0:x0INy1}&Az0:z0INx1==z0INy1’))),
               substitute((P[2]),parse(’x1’),substitute((P[3]),parse(’y1’),parse(’x1={x0:>x1‘x0}&y1={x0:>y1‘x0}&Az0:x1‘z0=y1‘z0’)))],’’]]

    if (P[0]==’unary’ and P[2][0]== ’unary’ and P[2][1]==[’operator’,’THE’]):
         C1=substitute(P[2][2],parse(’x1’),parse(’Ey0:Ax0:x0INx1==x0=y0’))
         C2=substitute(P[2][2],parse(’x1’),parse(’Ax0:x0INx1==x0=THEx1’))
         return [[[],[C1,P],’’],[[C2],[P],’’]]
    if (P[0]==’binary’ and P[3][0]== ’unary’ and P[3][1]==[’operator’,’THE’]):
         C1=substitute(P[3][2],parse(’x1’),parse(’Ey0:Ax0:x0INx1==x0=y0’))
         C2=substitute(P[3][2],parse(’x1’),parse(’Ax0:x0INx1==x0=THEx1’))
         return [[[],[C1,P],’’],[[C2],[P],’’]]
    if (P[0]==’binary’ and P[2][0]== ’unary’ and P[2][1]==[’operator’,’THE’]):
         C1=substitute(P[2][2],parse(’x1’),parse(’Ey0:Ax0:x0INx1==x0=y0’))
         C2=substitute(P[2][2],parse(’x1’),parse(’Ax0:x0INx1==x0=THEx1’))
         return [[[],[C1,P],’’],[[C2],[P],’’]]
    D=defexpand(P)
    if not P==D:  return [[[],[D],’’]]
    return(’fail’)

def leftrot(S):
    if len(S[1])==0:  return ’goal’
    return [[S[0],S[1][1:]+[S[1][0]],S[2]], ’goal’]

def leftrotE(S):
    if len(S[1])==0:  return ’goal’
    if len(S[1])==1:  return ’goal’
    return [[S[0],[S[1][0]]+S[1][2:]+[S[1][1]],S[2]],’goal’]

def leftrot2E(S):
    if len(S[1])==0:  return ’goal’
    if len(S[1])==1:  return ’goal’
    return [[S[0],[S[1][0]]+[S[1][-1]]+S[1][1:-1],S[2]],’goal’]

def Getleft():  act2(leftrot)

def GetleftE():  act2(leftrotE)

def Getleft2E():  act2(leftrot2E)

def rightrot2(S):
    if len(S[2])==0:  return ’goal’
    return [[S[0],S[1],[S[2][-1]]+S[2][0:-1]],’goal’]

def Getright2():  act2(rightrot2)

def leftrot2(S):
    if len(S[1])==0:  return ’goal’
    return [[S[0],[S[1][-1]]+S[1][0:-1],S[2]], ’goal’]


def Getleft2():  act2(leftrot2)

def rightrot(S):
    if len(S[2])==0:  return ’goal’
    return [[S[0],S[1],S[2][1:]+[S[2][0]]],’goal’]

def Getright():  act2(rightrot)

thecutterm=[]

def cut(S):
    global thecutterm
    return [[makesequent(S[1],[thecutterm]+S[2],’’),’goal’],[makesequent([thecutterm]+S[1],S[2],’’),’goal’]]

def Cut(t):
    global thecutterm
    T=parse(t)
    if not type(T)==’prop’:  return ’error’
```

```
    if not freevars(T)==[]:  return 'error'
    thecutterm=parse(t)
    act(cut)

# equality command

converse=False
inleft=False

def Converse():
    global converse
    converse=True
def Direct():
    global converse
    converse=False
def Inleft():
    global inleft
    inleft=True
def Inright():
    global inleft
    inleft=False

def equal(thesub,S):
    global converse
    global inleft
    if len(S[1])==0:  return 'goal'
    if not S[1][0][0]=='binary':  return 'goal'
    if not S[1][0][1]==['operator','=']:  return 'goal'
    if converse:
        b=S[1][0][2]
        a=S[1][0][3]
    if not converse:
        a=S[1][0][2]
        b=S[1][0][3]
    if inleft:
        if len(S[1])==1: return 'goal'
        return [[makesequent([S[1][0]]+[thesub(a,b,S[1][1])]+S[1][2:],S[2],""),'goal']]
    if not inleft:
        if len(S[2])==0:  return 'goal'
        return [[makesequent(S[1],[thesub(a,b,S[2][0])]+S[2][1:],''),'goal']]

def equal0(S):  return equal(strongsubstitute,S)
def equal1(S):  return equal(strongsubstitute1,S)
def equal2(S):  return equal(strongsubstitute2,S)

def Equal0():  act(equal0)
def Equal1():  act(equal1)
def Equal2(): act(equal2)

def Equaldr0():
    Direct()
    Inright()
    Equal0()

def Equaldl0():
    Direct()
    Inleft()
    Equal0()

def Equalcr0():
    Converse()
    Inright()
    Equal0()

def Equalcl0():
    Converse()
    Inleft()
    Equal0()

def Equaldr1():
    Direct()
    Inright()
    Equal1()

def Equaldl1():
    Direct()
    Inleft()
```

```
    Equal1()

def Equalcr1():
    Converse()
    Inright()
    Equal1()

def Equalcl1():
    Converse()
    Inleft()
    Equal1()

def Equaldr2():
    Direct()
    Inright()
    Equal2()

def Equaldl2():
    Direct()
    Inleft()
    Equal2()

def Equalcr2():
    Converse()
    Inright()
    Equal2()

def Equalcl2():
    Converse()
    Inleft()
    Equal2()



# We are ruling that all indexed constants and instantiables
# are proof-generated objects.  They will be indexed to manage newness
# and the instantiation rules.

# the definition facility is important for the style of proof being
# taught.  On the other hand, I also want "general" objects that can be
# used in theorem matching.

#when a quantifier rule is executed, we can take the string name of the
#witness or arbitrary object to be the string name of the variable, then
#give that a suitable index.  No hazard ensues, because the indices are new
#each time, and it should make proofs more readable.

# the next essential addition is quantifier rules.   For this I need
# constant and instantiable index management and global substitution for
# instantiables.

# the Done command already needs more matching to recognize identity up
# to alpha conversion.

# rules for abstractions are not essentially more difficult than the quantifier rules.

# the definition facility is vital to teaching the proof style  Im interested in.

# after that would be theorem application.  For this I need matching.
# Really fancy matching that does instantiation on the fly would be fun.
# for constants, we can prove theorems with indexed constants in them to
# get generality.  This is quite compatible with the regular use of these
# in quantifier proofs, anyway.  Operators with generality functions have to be declared.

# fancy equality rules are very sophisticated.

def toreturn(s):
    if s == '':  return s
    if s[0]=='\n':return ''
    return s[0]+toreturn(s[1:])

def printproof(P):
    print(printsequent(P[0]))
    if len(P)<2:
        print ('bad proof structure')
        return 0
```

```
    if P[1]=='goal': print('\ngoal\n')
    if completeproof(P):  print('proved')
    if not completeproof(P):  print('not yet proved')
    if P[1]==[]: print('\ntrivial\n')
    if not P[1]=='goal' and not P[1]==[]:
        bystring=''
        for Q in P[1]: bystring=bystring+'\nAND\n'+toreturn(str(Q[0][0]))
        bystring='by '+bystring[5:]
        print(bystring+'\n')
        for Q in P[1]:  printproof(Q)

# set up for global instantiation

# returns maximum object index

def maxindex(t):
    global theproof
    if (t[0]=='constant' or t[0]=='instantiable') and len(t)==3:  return int(t[2])
    if t[0]=='unary':  return (maxindex(t[2]))
    if t[0]=='binary':
        m=maxindex(t[2])
        n=maxindex(t[3])
        if m>n: return m
        return n
    return -1

def maxlistindex(L):
    if len(L)==0:  return -1
    m=maxindex(L[0])
    n=maxlistindex(L[1:])
    if m>n: return m
    return n

def maxsequentindex(S):
    m=maxlistindex(S[1])
    n=maxlistindex(S[2])
    if m>n: return m
    return n

def listsub(t,v,L):
    if L==[]:  return []
    global theproof
    return [substitute(t,v,L[0])]+listsub(t,v,L[1:])

def sequentsub(t,v,S):
    global theproof
    return [S[0],listsub(t,v,S[1]),listsub(t,v,S[2])]

def proofsub(t,v,P):
    global theproof
    if P[1]=='goal':  return [sequentsub(t,v,P[0]),'goal']
    return [sequentsub(t,v,P[0]),prooflistsub(t,v,P[1])]

def prooflistsub(t,v,L):
    global theproof
    if L==[]:  return []
    return [proofsub(t,v,L[0])]+prooflistsub(t,v,L[1:])

def instantiate(t,v,P):
    global theproof
    if not v[0]=='instantiable':  return P
    if maxindex(t) == 'infinity' or (not maxindex(v) == 'infinity' and not maxindex(t)<maxindex(v)): return P
    return proofsub(t,v,P)


def proplistsub(t,v,L):
    if L==[]:  return []
    global theproof
    return [propsubstitute(t,v,L[0])]+proplistsub(t,v,L[1:])

def propsequentsub(t,v,S):
    global theproof
    return [S[0],proplistsub(t,v,S[1]),proplistsub(t,v,S[2])]

def propproofsub(t,v,P):
    global theproof
    if P[1]=='goal':  return [propsequentsub(t,v,P[0]),'goal']
```

```
    return [propsequentsub(t,v,P[0]),propprooflistsub(t,v,P[1])]

def propprooflistsub(t,v,L):
    global theproof
    if L==[]:  return []
    return [propproofsub(t,v,L[0])]+propprooflistsub(t,v,L[1:])

#  operator substitution into proofs

def oplistsub(t,v,L):
    if L==[]:  return []
    global theproof
    return [opsubstitute(t,v,L[0])]+oplistsub(t,v,L[1:])

def opsequentsub(t,v,S):
    global theproof
    return [S[0],oplistsub(t,v,S[1]),oplistsub(t,v,S[2])]

def opproofsub(t,v,P):
    global theproof
    if P[1]=='goal':  return [opsequentsub(t,v,P[0]),'goal']
    return [opsequentsub(t,v,P[0]),opprooflistsub(t,v,P[1])]

def opprooflistsub(t,v,L):
    global theproof
    if L==[]:  return []
    return [opproofsub(t,v,L[0])]+opprooflistsub(t,v,L[1:])



# this is the user command to select witnesses

def Inst(t,v):
    global theproof
    theproof=instantiate(parse(t),parse(v),theproof)
    shownextgoal(theproof)

# globally replace a propositional variable v with a sentence term t
# with no free variables

def PropInst(t,v):
    global theproof
    theproof=propproofsub(parse(t),parse(v),theproof)
    shownextgoal(theproof)

# globally replace a general (indexed) operator

def OpInst(t,v):
    global theproof
    theproof=opproofsub(getident(t)[0],getident(v)[0],theproof)
    shownextgoal(theproof)

def Showall():
    global theproof
    printproof(theproof)

# ran into serious difficulty with equality of lists in Python
# unheralded by the documentation; used a dirty fix using equality
# of printed forms.  In principle, this now implements first order logic,
# and bringing it in principle to NFU is not difficult -- add the
# basic rules for membership and equality.

# definitions and usable theorems would complete the tool.

#alpha equivalence, complete the effective definition of matches.

#pruning?  easy to add.  Automatic pruning is hard and should be added early
#because the data structure needs to be changed.

#Ability to globally replace general
#operators and propositional variables.  Operators should ideally be replaceable by
#abstractions directly.  Ive decided that indexed operators should always be
#general, just as in the case of constants and instantiables, but they must be
#declared nonetheless (and will have the same declared type and precedence
#as their parent if they are indexed versions of otherwise defined/declared operators).

#introduce new abstraction environments :: and ::> for abstraction using complex
```

```
#terms on left.  As in Marcel these have different definitions of free variable
#and substitution:  giving them different notation will mean that I do not have
# to either change the quite standard semantics of the simple construction or create
# an annoying exception.

#think about what hardwired behavior of pairing is wanted.

# note that because of very simple general structure of terms, Watson style
# navigation and rewriting might be easily added.

# the full basic logic of NFU is now installed in principle.  The logic of function application
# is not installed yet.  It isnt NFU + Infinity because there are no pairs yet.

#New feature:  the parser accepts all parenthesis pairs as equivalent, but always puts
#propositions in parentheses, abstractions in braces, and other objects in brackets

# relative precedences of propositional connectives, relations, and operations are now managed
# automatically:  there is a default floor precedence for each kind of operation
# (driven by the output type and the first input type):  the floors are automatically
# reset when precedences are inserted.  Precedences of unaries intended to be used as
# binders should be manually set to 0.  Precedences of abstraction operators should be 0.

# operators with strange mixed types might have odd precedence behavior (that is, ones
# with mixed proposition and object arguments)

# This completes my intended specification for the parser (except for new abstraction constructions).

#  further things needed in short term:  definition facility, logic of functions and application, description operator,
#  alpha-equivalence, cut rule, logic of pairing and projections,
#  theorem storage and use, global substitution for propositional variables and general operators, substitution rule for equality?

#  one conclusion mode added.   Think about adding constructive mode, too.  This one conclusion mode is better,
#  in that it has no effect on the underlying logic at all:  it is purely a modification of the display.
#  it may be worse in requiring more explanation.   Also added Getleft and Getright commands which rotate
#  the other way.  The starred negations *can* be explained honestly as negations of alternative
# conclusions.

# Definition facility is set up!

Theorems={}

Proofs={}

def Saveproof(name):
    Proofs[name]=theproof

def Loadproof(name):
    global theproof
    theproof=Proofs[name]

def getline(label,P):
    global theproof
    if not len(P)==2:  return 'bad'
    if not len(P[0]) == 3:  return 'bad'
    if toreturn(P[0][0])==label: return P
    if P[1]=='goal':  return 'bad'
    for Q in P[1]:
        F=getline(label,Q)
        if not F=='bad':  return F
    return 'bad'

def Savetheorem(label,name):
    global theproof
    if label in list(Theorems.keys()):  return 'cant get a theorem line'
    if name in list(Theorems.keys()):  return 'conflict'
    if name[:5]=='Line ':  return 'cant name a theorem as a line number'
    P=getline(label,theproof)
    if P=='bad':  return 'label not found'
    if completeproof(P):  Theorems[name]=P
    if not completeproof(P):  return 'not proved'

# this creates a sequent with proposition p as content
# and asserts it proved

def Axiom(name,p):
    global nextobjectindex
    nextobjectindex=1
```

```
    if not freevars(parse(p))==[]:  return 'bad'
    if not type(parse(p))=='prop': return 'bad'
    if not defsafe(parse(p)):  return 'bad'
    global nextsequent
    nextsequent=0
    global theproof
    theproof=[[name,[],[parse(p)]],[]]
    print(printsequent(theproof[0]))
    Savetheorem(name,name)

def singleton(p):  return [[],[p],'']

# distribution list for applying a sequent as a rule

def theoremexpand0(S):
    global thedifferential
    global oldobjectindex
    global nextobjectindex
    thedifferential=maxsequentindex(S)
    oldobjectindex=nextobjectindex
#    nextobjectindex=nextobjectindex+thedifferential+1
    return list(map(singleton,(list(map(instsubstitute0,S[1])))))+[[list(map(instsubstitute0,S[2])),[],'']]

def Thm(name):  return Theorems[name][0]

thetheorem=''

def UseThm0(S):
    global thetheorem
    return [[[thetheorem,Theorems[thetheorem][0][1],Theorems[thetheorem][0][2]],[]]]+distribute(theoremexpand0(Theorems[thetheorem][0]),S)

def Usethm(name):
    global thetheorem
    thetheorem=name
    act(UseThm0)

def Showdef(name):
    if not name in list(Definitions.keys()):  return 'error'
    if name in declaredconstants:
        print(name+':\n\n')
        print(termprint(['binary',['operator','='],['constant',name],Definitions[name]]))
        return 'done'
    if len(optypes[name])==3:
        print (name+':\n\n')
        print(termprint(['binary',['operator','='],['binary',['operator',name],['constant','.arg','1'],['constant','.arg','2']],Definitions[name]]))
    if len(optypes[name])==2:
        print (name+':\n\n')
        print(termprint(['binary',['operator','='],['unary',['operator',name],['constant','.arg','1']],Definitions[name]]))

def Showthm(name):
    if not name in list(Theorems.keys()):  return 'error'
    print (name+':\n\n')
    print(printsequent(Theorems[name][0]))

# enable commands in Marcel ML style

def l():  Left()

def r():  Right()

def gl(n):
    if n>1: Getleft(); gl(n-1)

def gr(n):
    if n>1: Getright(); gr(n-1)

def s(T):Start(T)

#  I believe this system has hit a critical mass.  It still needs things:
#  alpha equivalence in matching, the cut rule (though theorem cut
# is useful), the ability to load saved proofs, history management.

#  More powerful theorem use can be had with this approach if theorems
# include instantiables.  Fresh naming of variables in theorems and
#  increasing indices of instantiables when theorems are inserted would
#  be useful.
```

```
#  Global substitutions in proofs for general operators and for
#  propositional variables would be useful.

#  I am setting the default behavior to One Conclusion Mode for teaching.

Oneconclusion()

#  the propositional variable and general operator substitution procedures
#  are crucial for theorem usability.   The modifications of identifiers
# in theorems would be useful.  alpha conversion would be nice.  Substitution
# commands for equality use are important.

# the ability to globally replace a propositional variable or a general operator is now installed
# I have not installed the ability to replace an operator variable with an anonymous appropriately typed
# and possibly heterogeneous abstraction:  to do that one will need the definition facility for now.

# logic of functions!  cut!

# definitions of constants are needed -- this requires that we have declaration control in the type system
# for non indexed constants to prevent circularity.  We can have a list of declared constants:  this will handle the issue

# this approach differs from the ML approach in that our definition of substitution is very simple,
# and we have so far made no use of matching.  We have a very simple core.

# defined and declared constants are installed.  The type system will reject a non-indexed constant
# unless it has either been declared or defined.  Should I make an exception for numerals?

# have the expansion of functions also handled by defexpand.  Equality of functions should be
# much like equality of sets.

# installed the notion that the maximum index of a nonindexed instantiable is
# infinite.  This should support uses of instantiables in theorems.  It seems
# that the usefulness of this may be very limited.  The problem is that one cannot
# prove very much about infinite index instantiables.

# the idea of a data structure for heterogeneous abstractions (basically
# a sublist of the Define argument) presents itself:  one could make it
# easier to have a more general OpInst command.

# install cut.  install logic of functions.  Perhaps install alpha equivalence for matching.
# try writing some actual proofs.

#printing greatly improved.  Dots are no longer required, though they are allowed and the parser does use them internally.
#I might want the compact display as an alternative; I have preserved information about the original termprint function in comments.

#look through functions for error messaging and crash prevention

# definitely install some sort of equality rewriting, and the function logic
# must go in.

#Cut is installed

#installed alternative conclusion for equality on the right, which is quite complicated but solves problems I encountered in my demo proof:
#it checks for sethood of both terms after definitional expansion.  It now also
# checks for functionhood.  The moral, already seen in the previous version
# of Marcel, is that one should avoid hitting an equational conclusion with
# Right() if at all possible.

#also, fixed a bug in Define

# rewriting for equations is installed

# the logic of functions is installed

# How to build in a description operator?  By axiom it is easy.  But it would be nice to have  a notion
# of reduction. P(THEa) reduces to for some x, a={x} and P(a) OR for all x a is not {x} and P(emptyset)
# I could in principle install this for every basic context?

# a description operator is needed to be able to construct functions from sets.  It is interesting
# that the logic of functions is independent here.   The if then else term construction is enough
# to get infinity from the concept that every function is a set.

# figure out whether infinite index for unindexed instantiables is any use,
# and also whether infinite indexed expressions can instantiate infinite indexed
# variables safely (which would make it a bit more flexible).

# The definite description operator is installed.  No assumption at all
```

```
# is made about what THEa is when a is not a singleton set.  What happens is that if it
# notices a THEa subterm in a term not otherwise expanded, it introduces
# the hypothesis that a is a singleton in one sequent and the assumption
# that it is the singleton of THEa in the other.

# One could install the Hilbert symbol in the same way...

# It is very interesting that so far the logic does not commit one to Infinity.
#  The logic is full strength in that it can handle operators of any arity,
#  because it supports the Kuratowski pair.  The natural way for Infinity to
#  show up is to introduce the usual definition of a function as a set of ordered pairs.
#  The combination of this definition and the one-level type differential
# then forces a type level ordered pair.  The presence of the definite description
# operator is required to ensure that functions exist correlated with any functional set.
# The correlation between the +1 type functions which are primitive and the
# +3 type functional sets using the K pair is provable given definite description.

# It is also interesting that the logic here does not need to concern itself
# at all with elements of non-sets or values of non-functions.

# there remains the bug which increments variable indices by two instead of one.
# I ought to track that down.  It is really intractable, I cant figure it out.

# added notations proved and not yet proved to display of lines in printed proofs.
# separated two issues in my mind.  I think I do want ThmCut (in the old parlance,
# it will just be an extension of Usethm here).  So sequents should be provable as theorems,
# and mineable from the current proof; ThmCut is actually a device for avoiding manual
# matching of line numbers and very much in the spirit of the current version.

# Installed absurd symbol as conclusion in Oneconclusion() mode

# major and possibly buggy upgrades:  I have implemented saving and using general sequents as theorems.
# Savetheorem(sequent id, theorem name) saves sequent with given id (something like 'Line 3') as a theorem, with its proof,
# if it is proved.  Usethm(name) now acts like old ThmCut, including the transformation of constants to new instantiables.
# The Savetheorem command will not allow one to get a line whose name is a theorem name, nor will it allow a theorem
# to be named something like 'Line 1'.  I eliminated the infinite index idea for nonindexed instantiables; it would
# interfere with the Thmcut mechanism, and the Thmcut mechanism serves its intended purpose.

# things found in the ML version which are not found here:
#   1.  rewritefree  This was very useful and should be added.
#   1b.   Rewriting with biconditionals!
#   2.  matching?   The only place where matching occurs if one wants to use Thmcut instead of the old Usethm
#       is in the Done command.   Allowing Done to force instantiations is interesting.
#       In the old version, equality rules also used matching rather than identity:  this can
#       be considered here.  The old version of theorem application is not implemented here:
#       the new Usethm is the old Thmcut which I think is sensible.  Alpha equivalence
#       falls under this heading.
#   3.  automatic pruning.  This makes for much better-looking saved proofs.  I also ought to consider reordering sequents in
#       saved proofs to a more sensible order than the one they are dropped in.

#  things found in no version to add:  automatic Done as an option (only when there is literal identity).  Tactics like, apply Right over and over.
#  indexed propositional variables?  These might make propositional logic
# theorems more useful, especially if coupled with forced instantiation extended
# to prop variables?

# I should add the ability to recognize branching vs nonbranching rules
# which would enable application of things like "apply rights nonbranchingly
# as long as you can"  this may be possible to do imperatively in function
# innards by looking at the length of things generated by leftexpand
# and rightexpand.   Automatic Done is also good.  I should refamiliarize
# myself with pragmatics of the prover.

# logging of scripts to build proofs is probably wanted.

# a real aggravation in sample proofs is equality rule jumping the gun
# when equations are actually trivial by definition.  Could I do this
# by having equality right rule check for definition expansions
# and do *just* that by preference?  The hazard would be explosive definition expansion?
# Also,
# a variant on the left rule for sets looking like the right rule for sets
# might be desirable.  One could also
# have a dedicated definition expansion commands.

# made it so right rule for equality does definition expansion rather than the last
# resort

#Think about second order quantification.  This is workable (the changes
```

```
# to the parser are clear enough).  It would require instantiation with
# heterogeneous relations, which is probably no bad thing.  It would fit
# with the Frege application.  The semantics are also clear:  it makes
# the type structure a bit more complex.

# Think about strongly cantorian types and even NFU*.  At least Rosser's
# axiom of counting is needed for Frege.

#Oct 15 2015  OneConclusion in the style of the ML version is supported, invoked by the command Oneconclusion2()
#installed emulation of ML style commands  l() r() gl(n) gr(n) s
#changed numbering of conclusions presented as negative premises so that they are numbered
#(starred) as alternative conclusions.

# Project:  turn this into a full production version.

# what are the wow features of the ML version?

# 1.  autoprune.
#also, don't add extra steps to proofs
# on application of reordering or pruning.  second item is done.

# 2.  automatic instantiation as part of matching.  Matching
# pairs.  So we need pairs and projections as a primitive construction?

# 3.  rewritefree is really useful.

# 4.  complex terms on left side of abstractions.  With this
#  approach, this might require consideration of partial functions.

# wow feature not found in any version is creation of s.c. types.

# unstratified quantification exists in the ML version -- there is the
# question of how useful it is.  But one suspects one wants it.  And if one
# does, one might further want unstratified separation for s.c. sets.

# Oneconclusion2()

# install narrative comments on proofs.  This adds comments to the
# name components of sequents constructed.  It appears to be fairly easy
# to do with this version.  Comments are designed for one-conclusion mode.

# narrative comments are installed for propositional and quantifier logic.  They do presume a one conclusion format.
```