443

INTERNATIONAL COMPUTERS LIMITED
SYSTEMS STRATEGY CENTRE

REF: EB 83/6
DATE: 4th May 83

FINITE COMPUTATION PRINCIPLE
An Alternative Method Of Adapting Resolution For Logic Programming

E. Babb

ABSTRACT

Currently PROLOG implements resolution by means of symbolic substitution. The result is that **symbolic** operations (eg on lists) in PROLOG are reversible, whilst **data** operations (eg arithmetic) are not. This paper proposes an adaption to the resolution principle called the **Finite Computation Principle (FCP).** Using FCP, symbolic substitution is still available but is performed by a special predicate.

FCP gives the two important benefits of **Order independence** and **control over infinite processes.** In addition, FCP improves reversibility and simplifies the connection of logic to existing languages.

A logic language called Prolog M has been implemented using FCP. This provides standard negation, disjunction, conjunction, universal quantifiers and existential quantifiers. An important feature of the implementation is that if two Prolog M programs are **equivalent** according to the tautologies of Predicate Calculus, then these two programs will generate **identical** answers.

1. INTRODUCTION

During the 1970s the author was actively involved in the hardware and software design of the ICL Content Addressable File Store (CAFS). It was during this period that a method of making queries to a database without the reference to relation or file names [2] was proposed. This shorthand was made possible by including a limited mathematical model in the language interpreter. This model being made up of joins of relations. This technique has proved successful with database users but was limited to joins of physical relations - i.e. conjunctions of predicates. It was as a result of trying to generalise this model that it was realised how useful Prolog might be in this area.

Prolog is order sensitive. Despite the name, Prolog is not a true logic language and the database query below must be written in a particular order.

Manweight(x,w) , w < 20	**works!**
w < 20 , Manweight(x,w)	**Errors!**

In a database query, it is essential that the terms can be written in any order. Warren [7] recognises this in his CHAT80 database system. In CHAT80 additional features are included to allow order independence.

Prolog is very likely to go infinite. For example, Define **Append** in the standard way and then make the following queries:

Append(x,(4),w)	**prints infinite formula or infinite number of values**
Append((4),y,w)	**prints w = (4.y) or infinite list of values.**
Append((4),y,w) , y=w	**just does nothing!**

In this last example, Prolog is trapped in a silent contradiction. One predicate generating an infinite number of instances of the y and w variables, whilst the subsequent predicate **y = w** always fails. In a large machine with almost unlimited storage resources such an infinite contradiction could be a very expensive bug.

The Finite Computation Principle (FCP) makes two things possible:

True order independence
No infinite processes

Ordinary resolution is still available using a pattern matching predicate. However, FCP allows operations on lists or sets of data to be carried out more securely.

The power of FCP appears when it is used in recursive definitions. Thus, most of the paper is concerned with explaining the operation of a number of key examples - in particular **APPEND**. The paper then hints at what may be possible in the future.

1.1 Notation

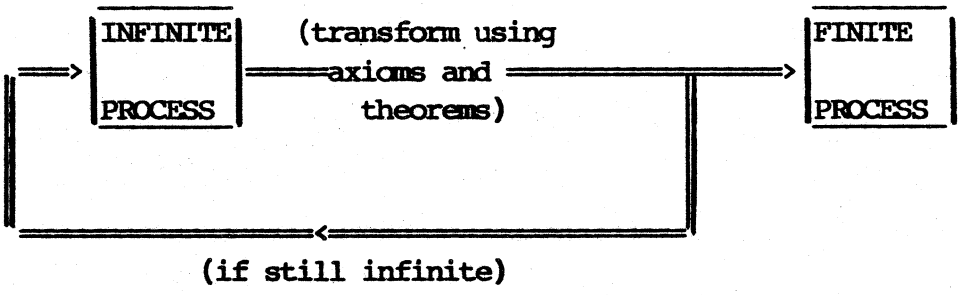
Prolog M uses a LISP like notation for predicates. However, for clarity this paper uses the conventional mathematical notation. Nevertheless, so that the flavour of Prolog M is not lost, the Prolog M syntax is often written along side in curly brackets.

Prolog M means Prolog with a Model[1,2]. It is hoped to describe the model aspect of Prolog M in a forthcoming issue of the ICL technical journal.

2. THE FINITE COMPUTATION PRINCIPLE (FCP)

The basic notion of the Finite Computation Principle is one that arises from the nature of logic programming. Some expressions written in a logic programming language may result in the infinite generation of data or some other endless process. FCP seeks to flag up infinite processes and put off their evaluation until the last possible moment by which time they may become finite processes as a result of information returned from other processes. **This is done by including in every built in predicate a test for the condition that makes it infinite.**

FCP detects that a process is infinite and then applies axioms and theorems to eventually create a finite process in the manner indicated below:



A process is either an atomic predicate, meta predicate (such as conjunction) or a user defined predicate. Current Prolog M only uses the axioms of logic to attempt to render a process finite.

To a limited degree CHAT80 uses something similar to FCP to delay the execution of negated predicates. The crucial feature about FCP is that every predicate should be able to identify the conditions that might make it infinite. It is then possible, as indicated above, to delay the execution of infinite predicates, even in recursive definitions.

In a fully working version of Prolog M, the optimal delaying of processes would also be included, as indeed it was in the ICL CAFS database system [3].

447

3. ATOMIC PREDICATES

Atomic predicates flag **infinite** if their use would generate an infinite solution set. For example:

x = 6	**FINITE:**	**only one solution**
x = y	**INFINITE:**	**(1,1)(2,2)(3,3)...**
2 = 4	**FINITE:**	**no solutions**
x = y + z	**INFINITE:**	**(1=1+0)(2=1+1)....**
(2,3) = u . x	**FINITE:**	**u is head of list ie 2 and x is tail of list ie (3)**

In Prolog M these have the syntax (=,x,6), (=,x,y), (=,2,4), (+,x,y,z), (.,(2,3),u,x) respectively.

Infinity is flagged by including in the definition of the predicate, code which will set an infinite flag for certain combinations of free variables.

4. LEFT TO RIGHT PREDICATE

Predicates are normally executed from left to right:

w < 20 , Manweight(x,w) { ((<,w,20)(Manweight,x,w)) }

Provided both are finite then the whole expression is finite. In this case the first atomic predicate is infinite and so the whole expression is infinite.

448

5. CONJUNCTION

Conjunction allows the machine to apply the axioms of logic to determine a finite ordering. Thus, if we write the following:

w < 20 & Manweight(x,w) { (&(<,w,20)(Manweight,x,w)) }

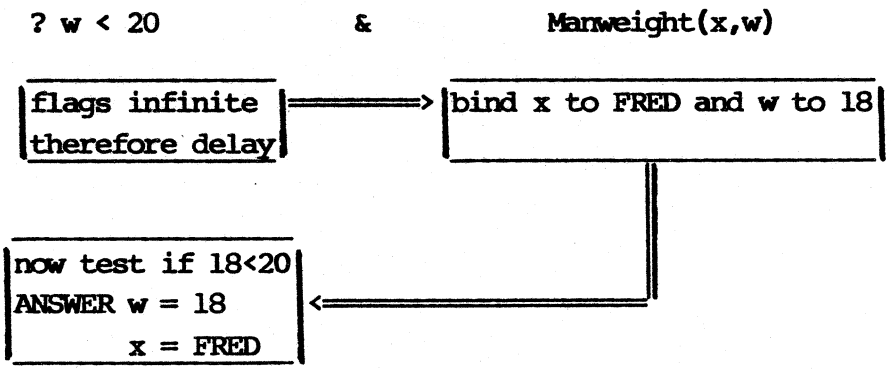
The machine will attempt execution from left to right:

w < 20 , Manweight(x,w)

The result is infinite and so using the axiom **A & B <-> B & A** the reverse ordering is tried:

Manweight(x,w) , w < 20

If we assume manweight has instance FRED,18 then execution is as follows:



If there had been nested conjunctions, these would be collapsed down so that **((A & B) & C)** would be replaced by **(A & B & C)**.

6. DISJUNCTION

Disjunctions of two or more predicates are executed as two quite separate processes. Thus:

```
(x = 3 or x = 4) & f(x)          { (&(or(=,x,3)(=,x,4))(f,x)) }
```

is executed as two processes:

```
x = 3 & f(x)                     { (&(=,x,3)(f,x)) }
x = 4 & f(x)                     { (&(=,x,4)(f,x)) }
```

First, x is given the value 3 and f is executed. Second x is given the value 4 and f is again executed. For a disjunction to be finite both these processes must be finite.

7. EXISTENTIAL QUANTIFICATION

If there exist values of x1,x2,... that satisfy an expression p then the expression q is executed:

```
some(x1,x2,...)(p) , q           { (some (x1,x2,...)p) }
```

We evaluate this by forcing x1,x2,.. to be variables local to p. Thus, they start off initially as free variables. If there are instances of these variables locally in p then the expression q is executed.

8. NEGATION

Negation is implemented by transforming the negated expression so that the **not** is moved to a subexpression using one of the three axioms:

not(p&q) —> notp or notq	{ (not(&,p,q)) --> (or(not,p)(not,q)) }
not(p or q) —> notp & notq	{ (not(or,p,q))--> (&,(not,p)(not,q)) }
notnotp —> p	{ (not(not,p))-->p }

Eventually, the expression cannot be changed because none of these transforms can be applied. It will then be found that p is either an atomic predicate or an existential quantifier. We therefore actually execute the **not(p)** predicate. The **not** means no instances. Thus, the **not** is executed by checking that the predicate p has indeed no instances. If this is the case, then we allow execution of any statements that follow. This is "Negation as Failure" [4]. In the example:

not6=7 & x = 9 { (&(not(=,6,7))(=,x,9)) }

six does not equal seven, there is no instance, and so the next term x=9 is executed.

Negation has its own special infinities. A negation is finite only if all the free variables of p are externally bound. Thus, the free variable of the expression **x = 6** is x. Therefore, for **not x = 6** to be finite, x must be bound at the time when the **not** is executed. Clearly, if x had been free then there would be an infinite number of x values not equal to six. Again by trapping this infinite case it is possible for other processes to bind x.

8.1 SPECIAL CASE

Suppose an expression **not p** has free variables and is therefore infinite. It is sometimes possible, when the expression p starts with the quantifier **some**, to manipulate **p** to give a new expression **p'** which generates the bindings for these free variables. The resulting expression **p'&not p** now being finite. For example, the infinite statement

```
not some(x)(r(x)&not h(x,y))
                    {(not(some(x)(&,(r,x)(not(h,x,y))))) }
```

can be rendered finite by moving **h(x,y)**, the negated term in p, outside:

```
some(x)h(x,y)  &  not some(x)(r(x)&not h(x,y))
```

This new term now creates a finite set of bindings for y. A general theorem for transforming p to p' is given in reference [1].

9. UNIVERSAL QUANTIFIERS

Universal quantifiers are equivalent to negative existential quantifiers and so they are transformed before execution using the axiom:

```
all (x1,x2,...)(p) --> notsome(x1,x2,...)(not p)

                    { (all(x1,...)p)-->(not(some(x1,..)(not p))) }
```

452

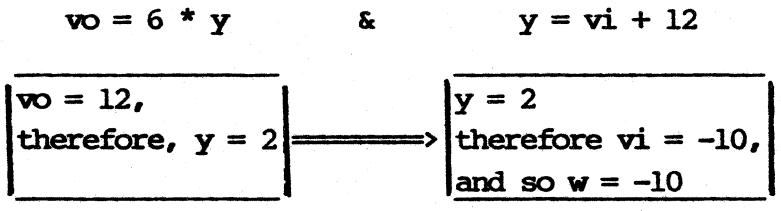
10. DEFINITIONS

Definitions allow complex expressions to be represented by a single predicate. Consider the definition:

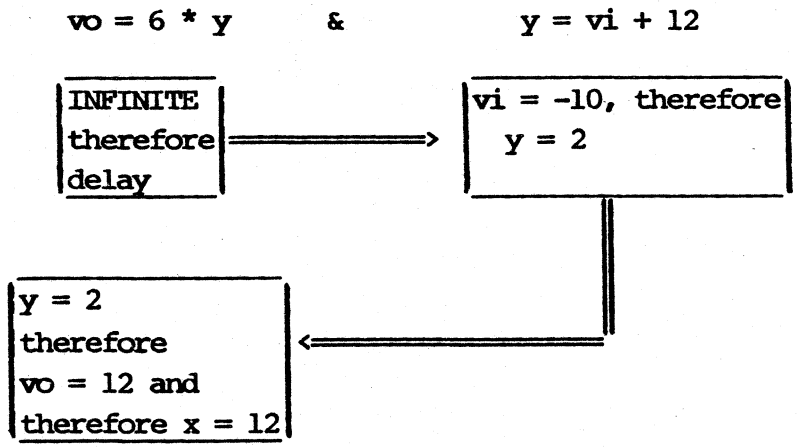
amplifier(vo,vi) <- vo = 6 * y & y = vi + 12

When this is called using the query ?amplifier(12,w), the variable vo inside the definition takes on the value 12 while the variable vi points to an identical location in store to w and hence become equivalent. The variable y is local to the definition and so there is an implicite existential quantifier.

This definition is fully reversible, so we can either ask the question **?amplifier(12,w):**



or the reverse question **?amplifier(x,-10):**



We can now define another predicate representing two amplifiers in cascade and still have reversibility:

amps(vo,vi) <- amplifier(vo,x) & amplifier(x,vi)

Term 1 or 2 being automatically selected depending whether vo or vi is bound.

10.1 RECURSIVE DEFINITIONS

Consider the operation of **append**. This can be defined by the single recursive definition which appends list x to list y to give list z:

```
append(x,y,z) <-    x = ()  &  y = z   or
                    x = u.x' & z=u.z' & append(x',y,z')
```

This definition states that if x is an empty list then lists y and z are equal. Otherwise, if we strip u off lists x and z then the remaining lists x' and z' are related by the append predicate. When used in recursion neither **or** nor **&** are order independent. This is because recursive calls to the append predicate always have the possibility of being infinite and so should always be written last. It may be sensible in some future implementation to automatically place such recursive calls last thus restoring order independence.

Using FCP this definition gives the following results. Readers interested in the details of the execution are refered to the appropriate appendix.

?append((2),(3),z)	**z = (2,3)**	**APPENDIX 1**
?append(x,y,(2,3))	**x = () y = (2,3)**	
	x = (2) y = (3)	**APPENDIX 2**
	x = (2,3) y = ()	
?append(x,(2),z)	**infinite flag set**	**APPENDIX 3**
?append((2),y,z)	**infinite flag set**	

Notice how FCP correctly traps the infinite process. Contradictions as mentioned earlier can therefore be trapped before execution:

```
?append((2),y,z) & y = z
    flags infinite
```

Without this facility a naive user could be faced with some expensive computer bills!

10.2 EXPLICITE TRAPPING OF INFINITE PROCESSES

Suppose we define a factorial predicate **fact'(x,n)** which gives the factorial x of a number n. When x and n are both free we find that **fact'** executes an **actual** infinite loop. To prevent this infinite loop we precede **fact'** by the predicate **free:**

fact(x,n) <- free(x,n) , fact'(x,n)

The predicate **free** flags infinite if all its arguments are free. Thus by detecting that x and n are both free, factorial is now secure against infinite loops.

Notice that **append** did not require any such trap to stay finite.

11. THE FUTURE

Prolog M uses the axioms of logic to transform an infinite expression to a finite expression. However, the capabilities of the language could be considerably extended if the user were also able to define his own infinite to finite axioms and theorems. Below is a simple example of an axiom to allow a natural way of writing a range of numbers:

x > xmin & x < xmax & integer(x) <- range(x,xmin,xmax)

{ (define (&(>,x,xmin)(<,x,xmax)(integer,x)) (range,x,xmin,xmax))}

It is now possible to write the query:

?x > 1 & x < 12 & integer(x)

and obtain the integers 2,3,...10,11 using the range predicate.

We can define new functions in the same way that we can bind variables to values. For example:

quadfn(x) = "x*x + 2*x - 4"

{(=,quadfn,'(lambda(x)(plus(times,x,x)(times,2,x),-4)))}

binds the function variable **quadfn** to **"x*x + 2*x - 4"** using a lambda expression. The function quadfn can then be used in an equality predicate:

y = quadfn(x) { (=,y,(quadfn,x)) }

Unlike normal equality, this is finite only if **x** is bound.

Predicates are often defined in terms of a forward and reverse function. A reversible quadratic function **quad(y,x)** is defined as the conjunction of **quadfn** and **quadreversefn.** The appropriate function being chosen by FCP.

12. CONCLUSION

Prolog M is still in its infancy. There are at least three important questions left unanswered:

1. By trapping the generation of infinite formulas, will FCP make conventional resolution more flexible?

2. Can trace facilities easily explain why programs are infinite?

3. Can we easily include user defined theorems?

ACKNOWLEDGEMENTS

The author is grateful to the late Roy Mitchell, Vic Maller, Norman Truman, Martin Stears and the other members of the ICL Systems Strategy Centre, Stevenage who have helped to formulate and develop the ideas in this paper.

REFERENCES

[1] BABB E — SYSTEM MODELLING LANGUAGE (SML) For Enquiries to a Business or Scientific Model. ICL Technical Note TN 82/1, 1982

[2] BABB E — Joined Normal Form: A storage encoding for relational databases. ACM Trans. on Database Systems. December 1982

[3] BABB E — Implementing a Relational Database by means of Specialised Hardware. ACM TODS, June 1979

[4] CLARK K & TARNLUND S.A.(Eds) — Logic Programming, Academic Press, 1982

[5] CLOCKSIN W & MELLISH C — Programming in PROLOG, Springer-Verlag, 1981

[6] KOWALSKI R A — Logic For Problem Solving, North Holland 1979

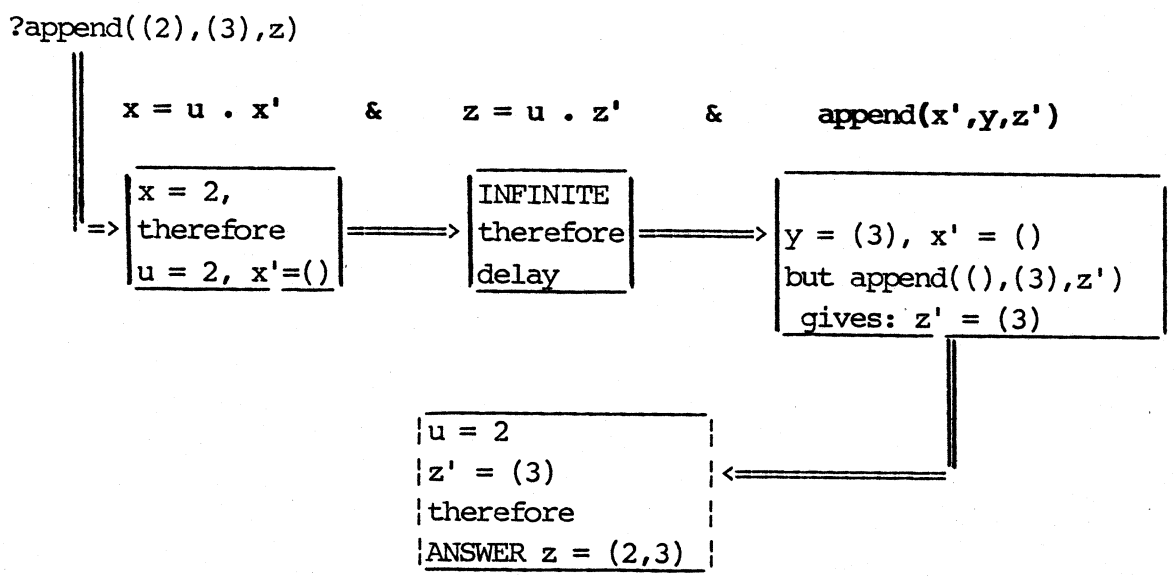
[7] WARREN D — Efficient Processing Of Interactive Relational Database Queries Expressed In Logic, Edinburgh DAI Research Paper No 156

[8] WARREN D & PEREIRA F — An Efficient Easily Adaptable System For Interpreting Natural Language Queries, Edinburgh DAI research paper no 155

457

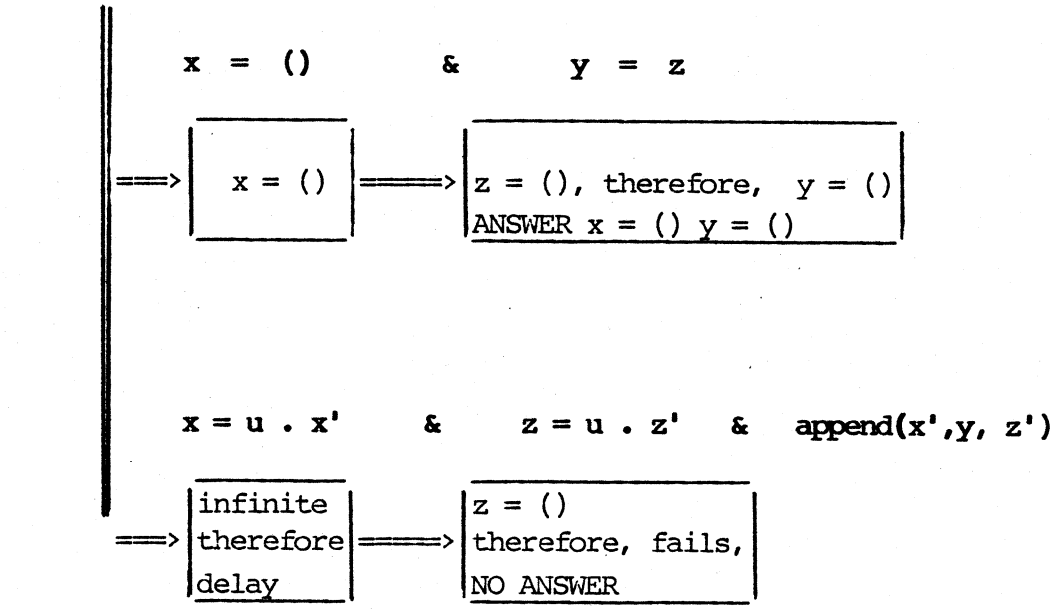
Appendix 1 Append forward

What is the result of appending (2) to (3)?



Appendix 2a Append backwards

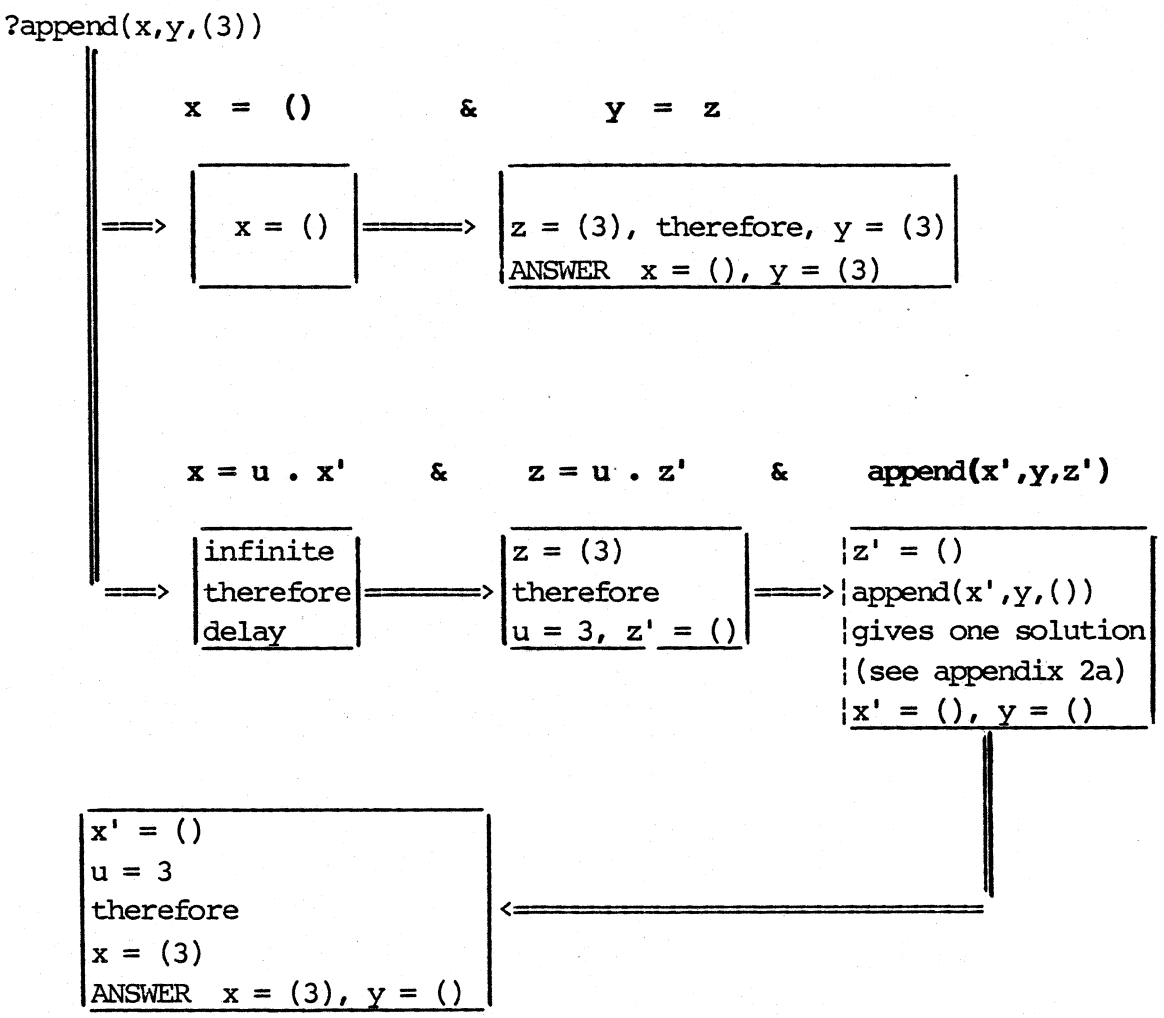
What two lists appended together give the empty list?



458

Appendix 2b

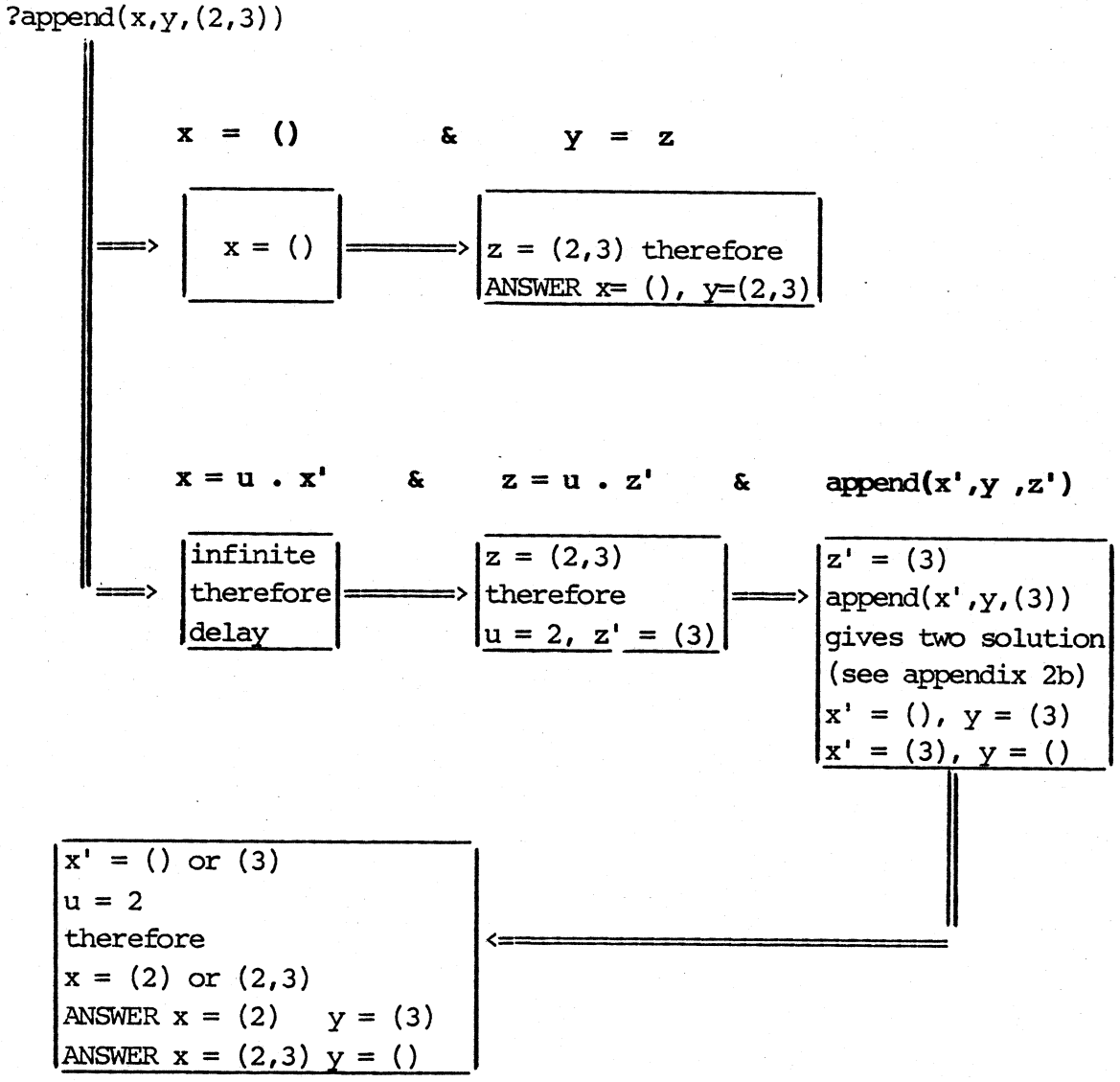
What two lists appended together give the list (3)?



459

Appendix 2c

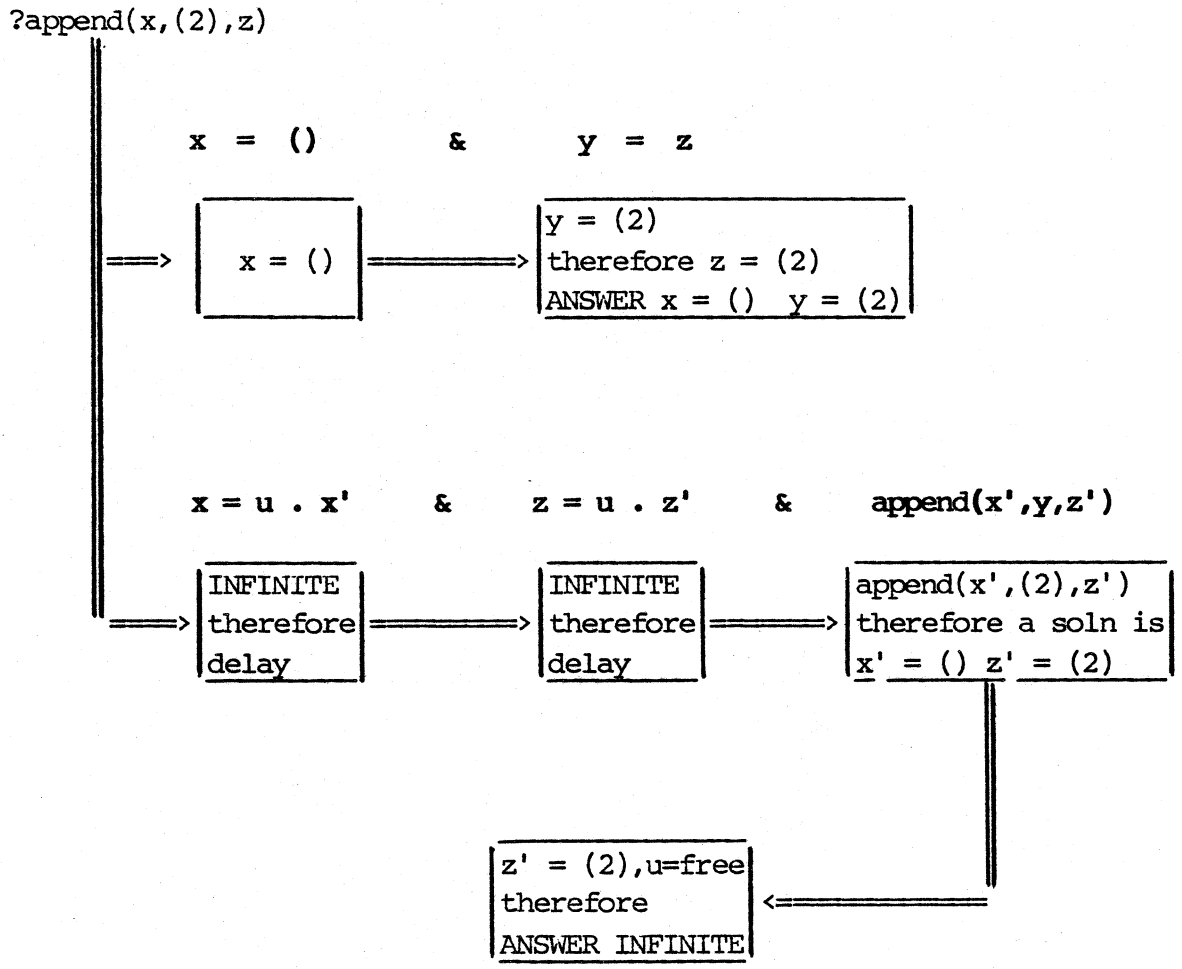
What two lists appended together give the list (2,3)?



460

Appendix 3 Append infinite

What are all the lists which end with a 2?



EB/SAC
SALLY836:EB 83/6