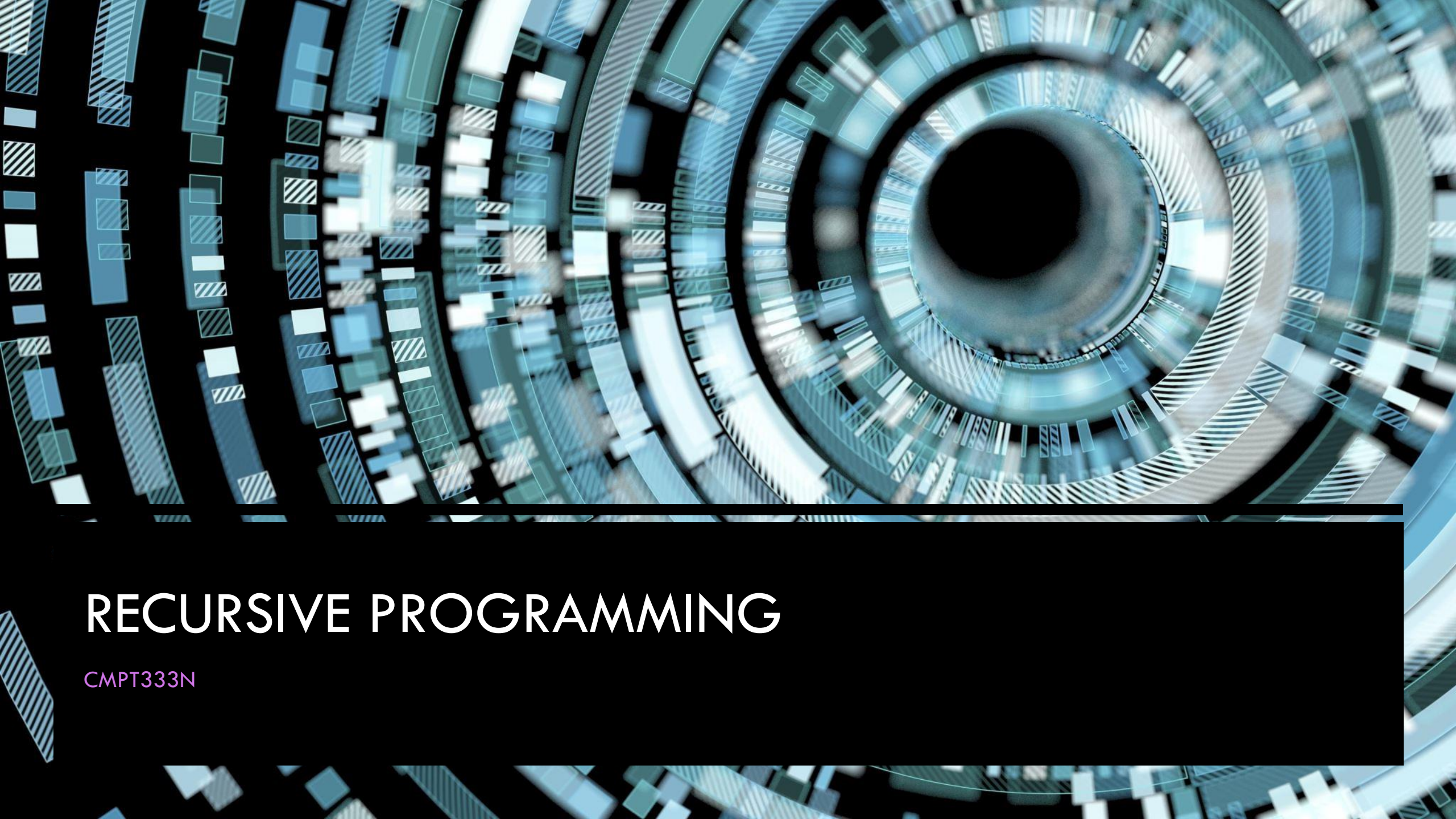


RECURSIVE PROGRAMMING

CMPT333N

ARITHMETIC

The simplest recursive data type, natural numbers, arises from the foundations of mathematics.

Arithmetic is based on the natural numbers.

Operations of arithmetic are usually thought of functionally rather than relationally.

It is more natural to discuss the underlying mathematical issues, such as correctness and completeness of programs.

NATURAL NUMBERS

- The natural numbers are built from two constructs:
  - The constant symbol 0
  - The successor function **s** of arity 1
- All the natural numbers are then recursively given as

  `0, s(0), s(s(0)), s(s(s(0))),...`
- We adopt the convention that $\mathbf{s}^n(0)$ denotes the integer *n*, that is, *n* applications of the successor function to 0.

COMPLETENESS AND CORRECTNESS

- A program P is *correct* with respect to an intended meaning *M* if the meaning of *P* is a subset of *M*.
- It is *complete* if *M* is a subset of the meaning of *P*.
- It is complete and correct if its meaning is identical to M.
- Proving correctness establishes that everything deductible from the program is intended.
- Proving completeness establishes that everything intended is deductible from the program.

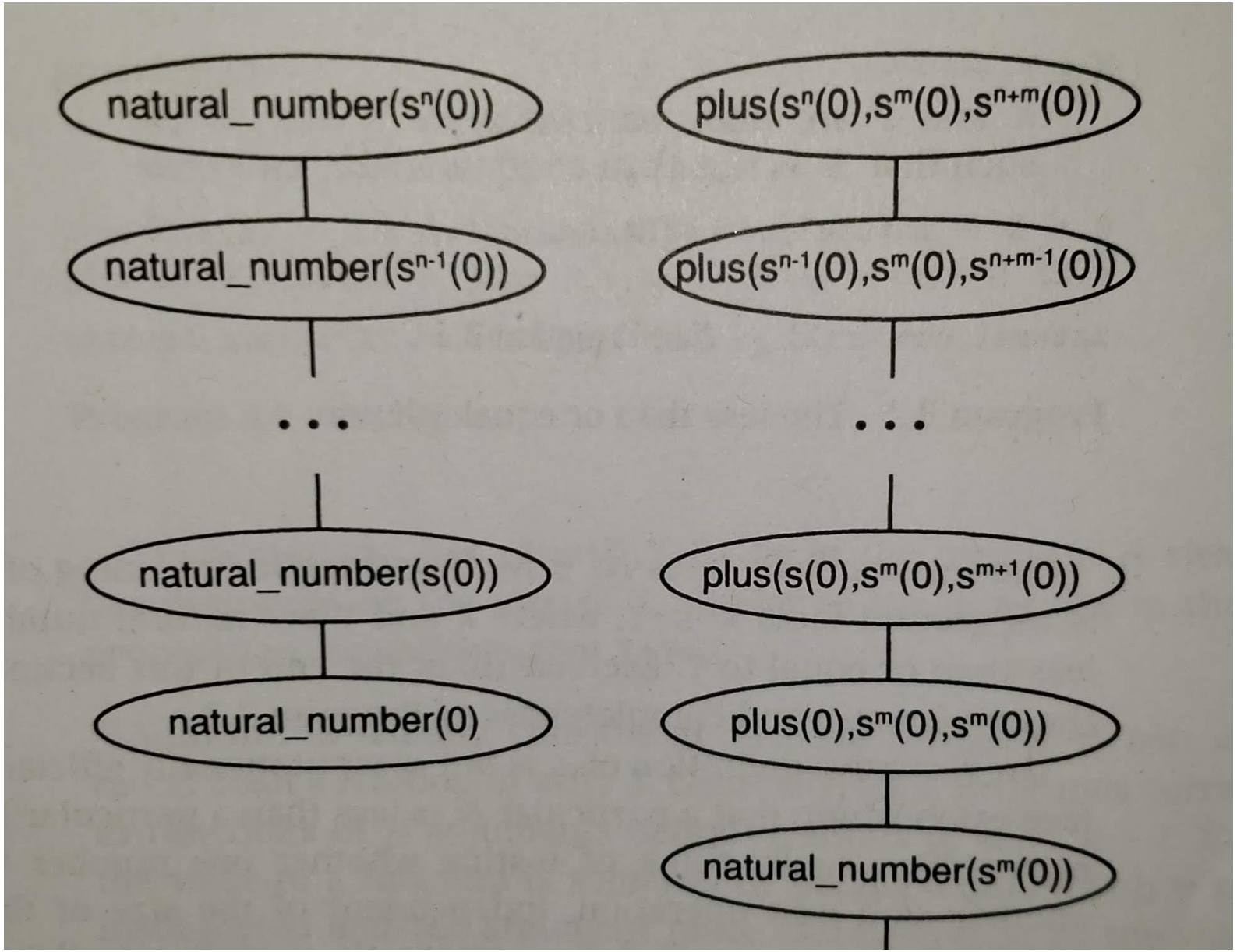
PROVING COMPLETENESS AND CORRECTNESS

- *natural_number*(X) ← X is a natural number.

```
natural_number(0).
natural_number(s(X)) ← natural_number(X).
```

- Proposition: this program is correct and complete with respect to the set of goals `natural_number(`s^i`(0))`, for i ≥ 0.
- Proof:
  1. Completeness.
     - Let n be a natural number.
     - We show that the goal `natural_number(n)` is deductible from the program giving an explicit proof tree. Either n is 0 or of the form s^n`(0)`.
     - The proof tree for the goal `natural_number(0)` is trivial.
     - The proof tree for the goal `natural_number(s(...s(0)...))` contains n reductions, using the rule in the program, to reach the fact `natural_number(0)`.

PROVING COMPLETENESS AND CORRECTNESS



PROVING COMPLETENESS AND CORRECTNESS

- Proof:

2. Correctness.
   - Suppose that `natural_number(X)` is deductible from the program in n deductions.
   - We prove that `natural_number(X)` is in the intended meaning of the program by induction on n.
   - If n = 0, then the goal must have been proved using a unit clause, which implies that X = 0.
   - If n > 0, then the goal must be of the form `natural_number(s(X'))`, since it is deductible from the program, and further, `natural_number(X')` is deductible in n-1 deductions.
   - By the deduction hypothesis, X' is in the intended meaning of the program, i.e., X' = $s^k(0)$ for some k $\geq$ 0.

ORDER OF NATURAL NUMBERS

- $X \leq Y \leftarrow$ X and Y are natural numbers, such that X is less than or equal to Y.

$0 \leq X \leftarrow$ `natural_number(X)` .

$s(X) \leq s(Y) \leftarrow X \leq Y$.

natural_number(X) ← (Previous definition)

- The natural numbers have a natural order.
- We denote the relation with a binary infix symbol, or operator, ≤, according to mathematical usage.
- The goal $0 \leq X$ has predicate symbol ≤ of arity 2, has arguments 0 and X, and is syntactically identical to '≤'(0,X).
- The relationship scheme is $N_1 \leq N_2$
- The intended meaning is all ground facts $X \leq Y$, where X and Y are natural numbers and X is less than or equal to Y.

ORDER OF NATURAL NUMBERS

- The recursive definition of $\leq$ is not computationally efficient.
- The proof tree establishing that a particular N is less than a particular M has M + 2 nodes.
- We usually think of testing whether one number is less than another as a unit operation, independent of the size of the numbers.
- Indeed, Prolog does not define arithmetic according to the axioms presented here but uses the underlying arithmetic capabilities of the computer directly.

ADDITION

plus(X,Y,Z) ← X, Y, Z are natural numbers such that Z is the sum of X, Y.

```
plus(0,X,X) ← natural_number(X).
```

```
plus(s(X),Y,s(Z)) ← plus(X,Y,Z).
```

- Addition is a basic operation defining a relation between two natural numbers and their sum.
- A recursive program captures the plus relation that we discussed previously in a more elegant and compact way.
- The intended meaning of the program is the set of facts `plus(X,Y,Z)`, where X,Y, and Z are natural numbers and X+Y=Z.

PROVING COMPLETENESS AND CORRECTNESS

- Proposition: The programs for natural numbers and addition constitute a correct and complete axiomatization of addition with respect to the standard intended meaning of `plus/3`.
- Proof:
  1. Completeness.
     - Let X,Y, and Z be natural numbers such that X+Y=Z.
     - We give a proof tree for the goal `plus(X,Y,Z)`.
     - If X equals 0, then Y equals Z.
     - Since the program for natural numbers is a complete axiomatization of the natural numbers, there is a proof tree for `natural_number(Y)`, which is easily extended to a proof tree for `plus(0,Y,Y)`.
     - Otherwise, X equals $s^n(0)$ for some *n*.
     - If Y equals $s^m(0)$ the Z equals $s^{n+m}(0)$.
     - Completeness is established in the right half of the tree is slide 6.

PROVING COMPLETENESS AND CORRECTNESS

- Proof:
  2. Correctness.
     - Let plus(X,Y,Z) be in the meaning.
     - A simple inductive argument on the size of X, like the one used in the previous proposition, establishes that X+Y=Z.

FUNCTIONAL VS RELATIONAL

- Addition is usually considered to be a function of two arguments rather than a relation of arity 3.
- Generally, logic programs corresponding to functions of n arguments define relations of arity n+1.
- Computing the value of a function is achieved by posting a query with n arguments instantiated and the argument place corresponding to the value of the function un-instantiated.
- The solution to the query is the value of the function with the given arguments.

FUNCTIONAL VS RELATIONAL

- To make the analogy clearer, we give a functional definition of addition corresponding to the logic program:

```
0+X = X.

s(X)+Y = s(X+Y).
```

- One advantage that relational programs have over functional programs is the multiple uses that can be made of a program.
- For example, the query `plus(s(0),s(0),s(s(0)))?` means checking whether 1+1=2.
- As for ≤, the program for `plus` is not efficient.
- The proof tree confirming that the sum of *N* and *M* is *N*+*M*+2 nodes.

FUNCTIONAL VS RELATIONAL

- Posing the query `plus(s(0),s(0),X)?`, an example of the standard use, calculates the sum of 1 and 1.
- However, the program can just as easily be used for subtraction by posing a query such as `plus(s(0),X,s(s(s(0))))?`.
- The computed value of X is the difference between 3 and 1, namely 2.
- Similarly, asking a query with the first argument un-instantiated, and the second and third instantiated, also performs subtraction.

MULTIPLE SOLUTIONS

- A more novel use exploits the possibility of a query having *multiple solutions*.
- Consider the query `plus(X,Y,s(s(s(0))))?`.
- It reads: “Do there exists numbers X and Y that add up to 3.”
- In other words, find a partition of the number 3 into the sum of two numbers, X and Y. there are multiple solutions.
- A query with multiple solutions becomes more interesting when the properties of the variables is restricted.
- There are two forms of restrictions:
  1. Using extra conjuncts in the query
  2. Instantiating variables in the query.

RESTRICTIONS IN MULTIPLE SOLUTIONS

- Assuming a predicate `even(X)`, which is true if X is an even number, the query `plus(X,Y,N),even(X),even(Y)?` gives a partition of N into two even numbers.
- The second type of restriction is exemplified by the query `plus(s(s(X)),s(s(Y)),N)?`, which insists that each of the numbers adding up to N is strictly greater than 1.

MULTIPLE USES

- Almost all logic programs have multiple uses.
- Consider the program for ≤, for example.
- The query `s(0) ≤ s(s(0))?` checks whether 1 is less than or equal to 2.
- The query `X ≤ s(s(0))?` finds numbers X less than or equal to 2.
- The query `X ≤ Y?` computes pairs of numbers less than or equal to each other.

SYMMETRIC ARGUMENTS

- The plus program is not unique.
- For example, the logic program:

```
plus(X,0,X) ← natural_number(X).
plus(X,s(Y),s(Z)) ← plus(X,Y,Z).
```

  has the same meaning as the previous program.
- Two programs are to be expected because of the symmetry between the first two arguments.
- A proof of correctness and completeness given previously applies to this program by reversing the roles of the symmetric arguments.
- The meaning of the program for plus would not change even if it consisted of the two programs combined.
- This composite program is undesirable, however.
- There are several different proof trees for the same goal.
- It is important both for runtime efficiency and for textual conciseness that axiomatizations of logic programs be minimal.

TYPE CONDITION

- We define a *type condition* to be a call to the predicate defining the type.
- For natural numbers, a type condition is a goal of the form `natural_number(X)` .
- Without this test, facts such as $0 \leq$ `a` and `plus(0,a,a)`, where `a` is an arbitrary constant, will be in the programs' meanings.
- Type conditions are necessary for correct programs.
- However, type conditions distract from the simplicity of the programs and affect the size of the proof trees.
- We might omit explicit type conditions from the example programs.

TIMES

- The relation scheme is **`times(X,Y,Z)`**, meaning X times Y equals Z.
- *times(X,Y,Z)* ← *X*,*Y*, and *Z* are natural numbers such that *Z* is the product of *X* and *Y*.

`times(0,X,0).`

`times(s(X),Y,Z)` ← **`times(X,Y,XY)`**,
`plus(XY,Y,Z).`

`plus(X,Y,Z)` ← as previously defined.

EXPONENTIATION

- Exponentiation is defined as repeated multiplication.
- *exp(N,X,Y)* ← *N*,*X*, and *Y* are natural numbers such that *Y* equals *X* raised to the power of *N*.

```
exp(s(X),0,0).

exp(0,s(X),s(0)).

exp(s(N),X,Y) ← exp(N,X,Z), times(Z,X,Y).

times(X,Y,Z) ← as previously defined.
```

FACTORIAL

- A definition of the factorial function uses the definition of multiplication.
- Recall that N! = N x N-1x … x 2 x 1.
- The predicate factorial(N,F) relates a number N to its factorial F.
- *factorial(N,F)* ← *F* equals *N* factorial.

```
factorial(0,s(0)).

factorial(s(N),F) ← factorial(N,F1),
times(s(N),F1,F).

times(X,Y,Z) ← as previously defined.
```

MINIMUM

- Not all relations concerning natural numbers are defined recursively.
- An example is determining the minimum of two numbers via the relation minimum(N1,N2,Min).
- *minimum(N1,N2,Min)* ← The minimum of the natural numbers *N1* and *N2* is *Min*.

minimum(N1,N2,N1) ← **N1** ≤ **N2**.

minimum(N1,N2,N2) ← **N2** ≤ **N1**.

N2 ≤ **N1** ← as previously defined.

REMAINDER

- Composing a program to determine the remainder after integer division reveals an interesting phenomenon – different mathematical definitions of the same concept are translated into different logic programs.
- The first program illustrates the direct translation of a mathematical definition, which is a logical statement, into a logic program.
- The program corresponds to the existential definition of the integer remainder: "Z is the value of X mod Y if Z is strictly less than Y, and there exists a number Q such that X = Q x Y + Z"
- *mod(X,Y,Z)* ← Z is the remainder of the integer division of *X* by *Y*.

```
mod(X,Y,Z) ← Z < Y, times(Y,Q,QY), plus(QY,Z,X).
```

- We can relate this program to constructive mathematics.
- Although seemingly an existential definition, it is also constructive, because of the constructive nature of <, plus, and times.
- The number Q will be explicitly computed by **`times`** in any use of **`mod`**.

REMAINDER

- In contrast, the second definition is recursive.
- It constitutes an algorithm for finding the integer remainder based on repeated subtraction.
- The first rule says that X mod Y is X if X is strictly less than Y.
- The second rule says that the value of X mod Y is the same as X – Y mod Y.
- The effect of any computation to determine the modulus is to repeatedly subtract Y from X until it becomes less than Y and hence is the correct value.
- *mod(X,Y,Z)* ← Z is the remainder of the integer division of *X* by *Y*.

```
mod(X,Y,X) ← X < Y.

mod(X,Y,Z) ← plus(X1,Y,X), mod(X1,Y,Z).
```

REMAINDER

- The mathematical function X mod Y is not defined when Y is zero.
- Neither of the two programs has a goal mod(X,0,Z) in its meaning for any values of X or Z.
- The test of < guarantees that.
- The computational model gives a way of distinguishing between the two programs.
- Given a particular X,Y, and Z satisfying mode, we can compare the sizes of their proof trees.
- In general, proof trees produced with the second program will be smaller that those produced with the first program.
- In that sense, the second program is more efficient.

ACKERMANN'S FUNCTION

- Ackermann's function is the simplest example of a recursive function that is not primitive recursion.
- It is a function of two arguments, defined by three cases:
  1. *ackermann(0,N) = N+1.*
  2. *ackermann(M,0) = ackermann(M-1,1).*
  3. *ackermann(M,N) = ackermann(M-1,ackermann(M,N-1)).*
- *ackermann(X,Y,A)* ← *A* is the value of Ackermann's function for the natural numbers *X* and *Y*.

```
ackermann(0,N,s(N)).

ackermann(s(M),0,Val) ← ackermann(M,s(0),Val).

ackermann(s(M),s(N),Val) ←
    ackermann(s(M),N,Val1), ackermann(M,Val1,Val).
```

GREATEST COMMON DIVISOR

- The final example is the Euclidean algorithm for finding the greatest common divisor of two natural numbers.
- It is a recursive program not based on the recursive structure of numbers.
- The relationship scheme is **`gcd(X,Y,Z)`**, with intended meaning that Z is the greatest common divisor (or gcd) of two natural numbers X and Y.
- It uses either of the two previous programs for **`mod`**.
- *gcd(X,Y,Z)* ← Z is the greatest common divisor of the natural numbers *X* and *Y*.

```
gcd(X,Y,Gcd) ← mod(X,Y,Z), gcd(Y,Z,Gcd).

gcd(X,0,X) ← X > 0.
```

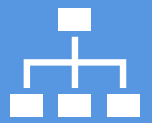
GREATEST COMMON DIVISOR

- The first rule is the logical essence of the Euclidean algorithm:
  - *The gcd of X and Y is the same as the gcd of Y and X mod Y.*
- A proof that the program is correct depends on the correctness of the above mathematical statement about greatest common divisors.
- The proof that the Euclidean algorithm is correct similarly rests on this result.
- The second fact is the base fact. It must be specified that X is greater than 0 to preclude `gcd(0,0,0)` from being in the meaning. The gcd of 0 and o is not well defined.

LISTS

The basic structure for arithmetic is the unary successor functor.

Although complicated recursive functions such as Ackermann's function can be defined, the use of a unary recursive function is limited.



This section discusses a binary structure, the *list*.

ELEMENT, HEAD, TAIL

- The first argument of a list holds an *element,* and the second argument is recursively the rest of the list.
- Lists are enough for most computations.
- Arbitrarily complex structures can be represented with lists.
- For lists, as for numbers, a constant symbol is necessary to terminate recursion.
- This “empty list,” referred to as nil, will be denoted here by the symbol [ ].
- We also need a functor of arity 2
- Historically, the usual functor for lists is “.” (pronounced dot), which overloads the use of the period.
- It is convenient to define a separate, special syntax.
- The term .(X,Y) is denoted [X|Y], X is called the *head* and Y is called the *tail*.

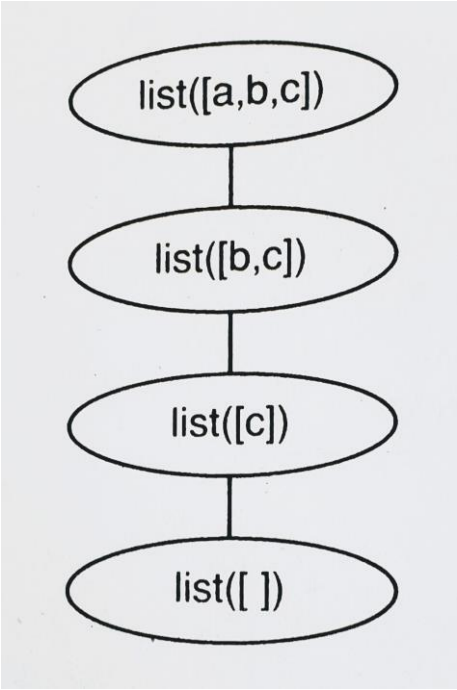
LIST DEFINITION

- The following program defines a list precisely.
- Declaratively is reads: “A list is either the empty list or a cons pair whose tail is a list.”
- The program is analogous to the one for natural numbers and is the simple type definition of lists.
- *list(Xs)* ← *Xs* is a list.

```
list([ ]).
list([X|Xs]) ← list(Xs).
```

PROOF TREE

- Below is the proof tree for the goal `list([a,b,c]).`
- Implicit in the proof tree are ground instances of the program, for example, `list([a,b,c]) ← list([b,c]).`



LIST MEMBER

- Because lists are richer data structures than numbers, a great variety of interesting relations can be specified with them.
- Perhaps the most basic operation with lists is determining whether a particular element is in a list.
- The predicate expressing this relation is **`member(Element,List)`**.
- *member(Element,List)* ← *Element* is an element of the list *List*.

```
member(X,[X|Xs]).

member(X,[Y|Ys]) ← member(X,Ys).
```

- Declaratively, this program is straightforward. X is an element of a list if it is the head of the list by the first clause, or if it is a member of the tail of the list by the second clause.
- The meaning of the program is the set of all ground instances **`member(X,Xs)`**, where **`X`** is an element of **`Xs`**.

SUBLISTS

- Our next example is a predicate **`sublist(Sub,List)`** for determining whether **`Sub`** is a sublist of **`List`**.
- A sublist needs the elements to be consecutive: [b,c] is a sublist of [a,b,c,d], whereas [a,c] is not.
- It is convenient to define two special cases of sublists to make the definition of **`sublist`** easier.
- It is good style when composing logic programs to define meaningful relations as auxiliary predicates.
- The two cases considered are initial sublists, or prefixes, of a list, and terminal sublists, or suffixes, of a list.

PREFIX AND SUFFIX

- The predicate **`prefix(Prefix,List)`** is true if **`Prefix`** is an initial sublist of a **`List`**,
- The companion predicate to **`prefix`** is **`suffix(Suffix,List)`**, determining if **`Suffix`** is a terminal sublist of **`List`**.
- For example, **`prefix([a,b],[a,b,c])`** is true and **`suffix([b,c],[a,b,c])`** is true.
- *prefix(Prefix,List)* ← *Prefix* is a prefix of *List*.

```
prefix([ ],Ys).

prefix([X|Xs],[X|Ys]) ← prefix(Xs,Ys).
```

- *suffix(Suffix,List)* ← *Suffix* is a suffix of *List*.

```
suffix(Xs,Xs).

suffix(Xs,[Y|Ys]) ← suffix(Xs,Ys).
```

DEFINING A SUBLIST

- An arbitrary sublist can be specified in terms of prefixes and suffixes: namely, as a suffix of a prefix, or as prefix of a suffix.
- *sublist(Sub,List)* ← *Sub* is a sublist of *List*.

a) Suffix of a prefix

```
sublist(Xs,Ys) ← prefix(Ps,Ys), suffix(Xs,Ps).
```

b) Prefix of a suffix

```
sublist(Xs,Ys) ← prefix(Xs,Ss), suffix(Ss,Ys).
```

c) Recursive definition of a sublist

```
sublist(Xs,Ys) ← prefix(Xs,Ys).
sublist(Xs,[Y|Ys]) ← sublist(Xs,Ys).
```

APPEND

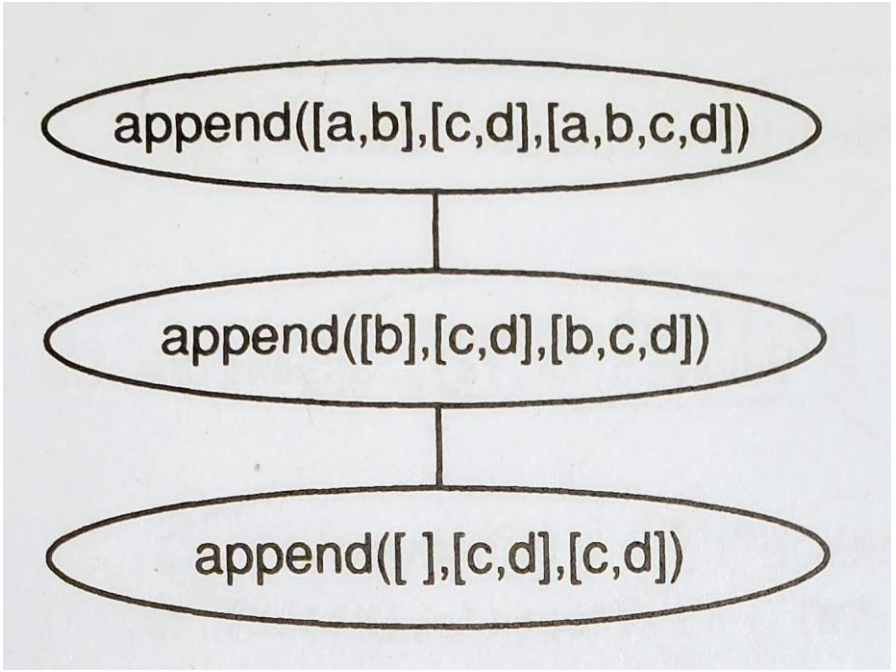
- The basic operation with lists is concatenating two lists to give a third list.
- This defines a relation, append(Xs,Ys,Zs), between two lists Xs, Ys and the result Zs of joining them together.
- The code for append is identical in structure to the basic program for combining two numbers.
- *append(Xs,Ys,XsYs)* ← *XsYs* is the result of concatenating the lists *Xs* and *Ys*.

```
append([ ],Ys,Ys).

append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).
```

PROOF TREE

- The proof tree for the goal `append([a,b],[c,d],[a,b,c,d])` is shown below.
- The tree structure suggests that its size is linear in the size of the first list.
- In general, if Xs is a list of n elements, the proof tree for `append(Xs,Ys,Zs)` has n+1 nodes.



USES OF APPEND

- There are multiple uses for append similar to the multiple uses for plus.
- The basic use is to concatenate two lists by posing a query such as `append([a,b,c],[d,e],Xs)?` With answer `Xs=[a,b,c,d,e]`.
- A query such as `append(Xs,[c,d],[a,b,c,d])?` finds the difference `Xs=[a,b]` between the lists `[c,d]` and `[a,b,c,d]`.
- Unlike plus, append is not symmetric in its first two arguments, and thus there are two distinct versions of finding the difference between two lists.

USES OF APPEND

- The analogous process to partitioning a number is splitting a list.
- The query **`append(As,Bs,[a,b,c,d])?`**, for example, asks for lists **`As`** and **`Bs`** such that appending **`Bs`** to **`As`** gives the list **`[a,b,c,d]`**.
- Queries about splitting lists are made more interesting by partially specifying the nature of the split lists.
- The predicates **`member`**, **`sublist`**, **`prefix`**, and **`suffix`**, introduced previously, can all be defined in terms of append by viewing the process as splitting a list.

USES OF APPEND

- The most straightforward definitions are for prefix and suffix, which just specify which of the two split pieces are of interest:

```
prefix(Xs,Ys) ← append(Xs,As,Ys).

suffix(Xs,Ys) ← append(As,Xs,Ys).
```

- **Sublist** can be written using two **append** goals. There are two variants:

```
sublist(Xs,AsXsBs) ← append(As,XsBs,AsXsBs),
append(Xs,Bs,XsBs).

sublist(Xs,AsXsBs) ← append(AsXs,Bs,AsXsBs),
append(As,Xs,AsXs).
```

- **Member** can be defined using **append**, as follows:

```
member(X,Ys) ← append(As,[X|Xs],Ys).
```

- That says that **X** is a member of **Ys** if **Ys** can be split into two lists where **X** is the head of the second list.

ADJACENT, LAST

- A similar rule can be written to express the relation **`adjacent(X,Y,Zs)`** that two elements **`X`** and **`Y`** are adjacent in the list **`Zs`**:

```
adjacent(X,Y,Zs) ← append(As,[X,Y|Ys],Zs).
```

- Another relation easily expressed through append is determining the last element of a list.
- The desired pattern of the second argument to append, a list with one element, is built into the rule:

```
last(X,Xs) ← append(As,[X],Xs).
```

REVERSE

- Repeated applications of append can be used to define a predicate reverse(List,Tsil).
- The intended meaning of reverse is that Tsil is a list containing the elements in the list List in reverse order to how they appear in List.
- An example of a goal in the meaning of the program is reverse([a,b,c],[c,b,a]).

REVERSE

- The naïve version is the logical equivalent of the recursive formulation in any language: recursively reverse the tail of the list, and then add the first element at the back of the reversed list.
- *reverse(List,Tsil)* ← *Tsil* is the result of reversing the list *List*.

```
reverse([ ],[ ]).

reverse([X|Xs],Zs) ← reverse(Xs,Ys),
append(Ys,[X],Zs).
```

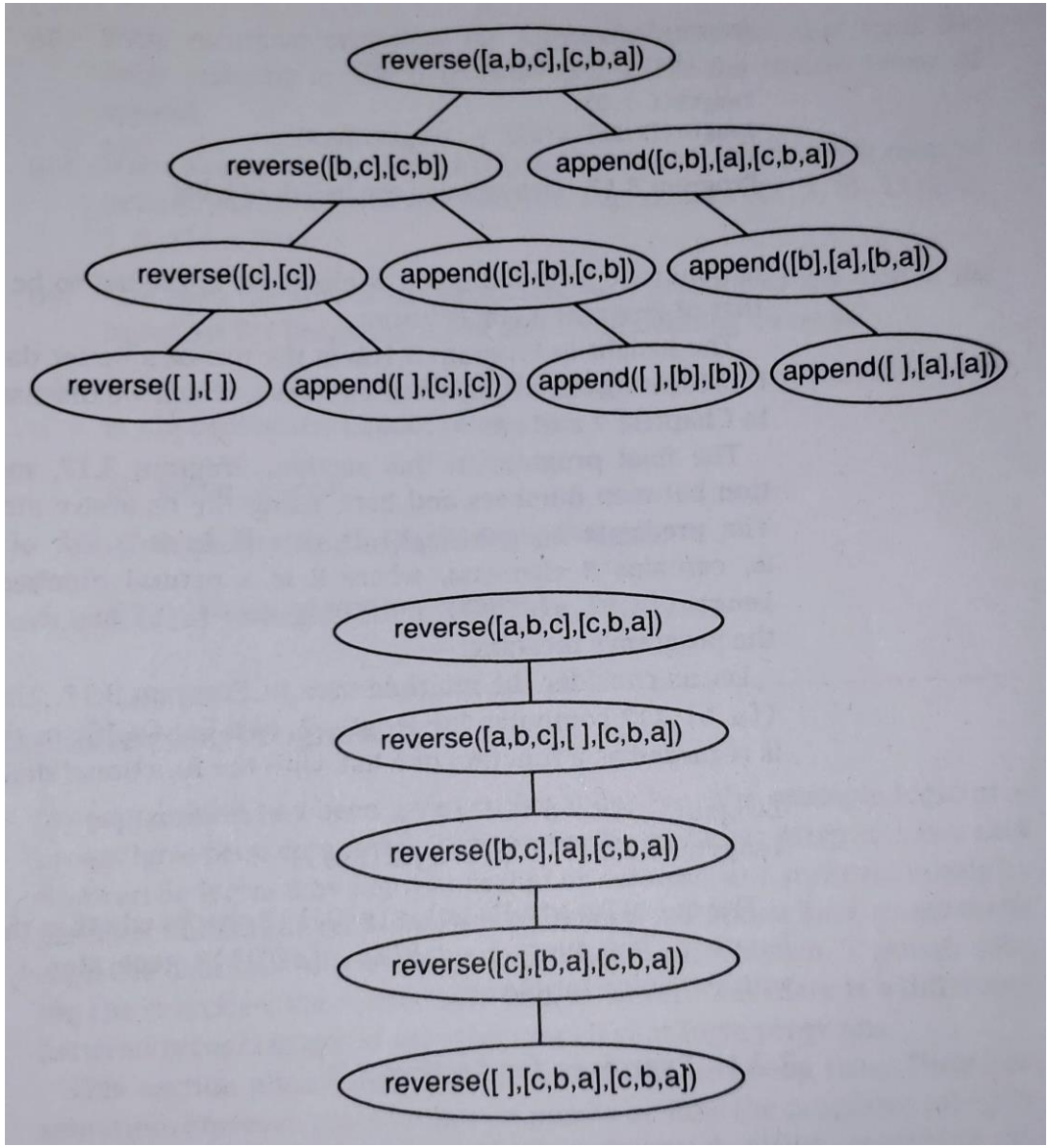
- There is an alternative way of defining **reverse** without calling **append** directly: We define an auxiliary predicate **reverse(Xs,Ys,Zs)**, which is true if **Zs** is the result of appending **Ys** to the elements of **X**s reversed.

```
reverse(Xs,Ys) ← reverse(Xs,[],Ys).

reverse([X|Xs],Acc,Zs) ← reverse(Xs,[X|Acc],Ys).

reverse([],Ys,Ys).
```

PROOF TREES FOR REVERSE



LENGTH

- The final program expresses a relation between numbers and lists, using the recursive structure of it.
- The predicate **`length(Xs,N)`** is true if **`Xs`** is a list of length **`N`**, that is, contains **`N`** elements, where **`N`** is a natural number.
- For example, **`length([a,b],s(s(0)))`**, indicating that **`[a,b]`** has two elements, is in the program's meaning.
- In this way, **`length`** is regarded as a function of a list, with the functional definition

```
length([ ]) = 0

length([X|Xs]) = s(length(Xs)).
```

- The query **`length([s,b],s(s(0)))?`** Checks whether the list [a,b] has length 2.
- The query **`length(Xs,s(s(0)))?`** Generates a list of length 2 with variables for elements.