In this course, we look at most topics from three different viewpoints. First we use the Story of Logic to motivate the ideas and ground them in a narrative that makes sense to every one interested in the subject. We also examine the subject that Hilbert called metamathematics where we use an informal mathematical theory to discuss properties of the object languages and logics we will study. These object logics are the topics that normally define the kind of logic course seen in courses and books. We will include among the object logics we study the Propositional Calculus (PC), First-Order Logic (FOL), Peano Arithmetic (PA), and others including at least one Programming Logic (PLog). Almost every college-level logic book covers PC and FOL under various names. One of the distinguishing features of logic as studied and taught by computer scientists is that the idea of a formal metalogic is important. The topic called logical frameworks in the CS literature is about that. Cornell graduates played a large role in defining this notion.

In this third lecture, we will develop further the metamathematics in Smullyan Chapter I, starting with his account of trees. My plan is to make more explicit the ideas from the textbook on trees (page 3) and to use symbolic notation for several of Smullyans definitions and theorems. This symbolic notation is standard in modern mathematics, and it will help introduce some of the logical notions we will study later in the object logics.

We will also develop a few algorithms in an informal notation that will eventually be formalized. In the informal metamathematics, we will be sensitive to whether our methods are constructive (computational) or not since this is one of the main themes we have examined in the Story of Logic (see Lecture 1 and part of Lecture 2); and it is a main theme in CS. The constructive approach will use algorithms in proofs and will make explicit algorithms that are implicit in proofs.

We start with the data type of potentially infinite trees over a discrete set of points.

1 Trees

1. Given a tree T, let $Points(T)$ be the type of the points in it. We use S for this type in these properties to connect with Smullyan's notation.

2. $eq(T)$ is a function that decides equality of elements in S, so it is a

computable function. We simply write eq when the tree T is clear. We declare its type as $eq : S \times S \to B$ where $S \times S$ is the product of S with itself whose elements are ordered pairs of elements of S, written as $< a, b >$ for a and b in S, and where B are the two Booleans, t for true and f for false.

3. $C0 : All\ x, y : S.(x = y \in S\ iff\ eq(x, y) = t)$. This property tells us that $x = y$ is the equality relation on the type S and that $x = y$ in S exactly when the eq function computes to the Boolean value t. We also express the computability idea by saying that the expression $eq(x, y)$ reduces to one of the two truth values in B.

4. $root(T)$ is a special point in S; it is also called the origin or root of the tree. We use the symbol $s1$ to abbreviate it, following the lecture.

5. $lev(T) : S \to \{x : N \mid x > 0\}$ declares a computable function from points to their level, a positive integer. We write simply lev when the tree T is clear in context. Smullyan uses a script lower case L for this function.

6. $C1 : All\ x : S.(lev(x) = 1$ iff $x = s1)$ or alternatively with more explicit declarations, $C1' : All\ T : Tree.All\ x : Points(T).(lev(T)(x) = 1$ iff $x = root(T)$ in $Points(T))$.

7. $Edge(T)$ is a relation that Smullyan calls R that relates two nodes when one is the successor of the other in the tree; $Edge(T) : S \times S \to Prop$ where $Prop$ is the type of all propositions. As explained in lecture, we use this type so that what we say in sensible to the constructive mathematicians and computer scientists as well as to the "classical mathematicians". This follows Bishop's idea to compromise if possible and be understood by a larger audience. For many mathematicians, $Prop$ is simply the type of Booleans, but this does not work constructively for computer science. When we write either $R(x, y)$ or following Smullyan, xRy, we say that "x is the predecessor of y" or "y is the successor of x".

8. Along with $Edge(T)$ we define $pred(T) : \{x : S \mid not(x = s1)\} \to S$ and we state $C2 : All\ y : \{z : S \mid not(z = s1)\}.(pred(y)Ry$ and $All\ x, y : S.(xRy \to x = pred(y)))$.

9. $C3 : All\ x, y : S.(xRy \rightarrow lev(y) = lev(x) + 1)$.

10. Smullyan also considers Ordered-Trees. These include a function $theta(T)$ whose type is $\theta(T) : S \rightarrow (N \rightarrow S)$. The junction points of S are those with a successor, and on these points $\theta(T)$ will produce the successors in order; that is, if θ abbreviates $theta(T)$, then $\theta(x, 1)$, $\theta(x, 2), \ldots$, are the successors of junction point x in order; these are distinct points if the number of successors is unbounded and otherwise, at the last successor, say k, we have $\theta(x, k) = \theta(x, m)$ for all $m > k$. The existence of such a k provides the number of successors of point x if there are finitely many. We agree that $\theta(x, 0)$ is x itself if x has no successors, so then and only then $\theta(x, n) = \theta(x, 0)$ for all positive n.

We call a point of a tree T an end point (of T) if it has no successors. Not every tree has an end point. To see this, define a path as any sequence of points beginning with the origin such that each point of the sequence, except the last if there is one, is the predecessor of the next, that is a sequence $s1, s2, s3, \ldots$ such that $si = pred(si + 1)$, and thus every point, except the last if there is one, has a successor, say $next(si) = si + 1$. If there is a last point of the path p, we denote it $last(p)$. A path is infinite iff every point in it has a successor. Clearly if every path in a tree T is infinite, then there are no end points in the tree. Define $Paths(T)$ as the set of all paths in the tree T. If p belongs to $Paths(T)$, then it has a last point iff it is finite. Define $FinitePaths(T)$ as the subtype of all paths which are finite. $Forallp : FinitePaths(T).last(p)$

Smullyan also defines simple points and junction points. He defines a maximal path or a branch to be a path which is either infinite or whose last point is a point with no successor in the tree, an end point of the tree (rather than of the path). Notice that there are trees with finite paths which are not branches because they end at an "interior point" of the tree. (Exercise, define an interior point of a tree. Does every tree have an interior point? Does every tree with at least two points have one? Does every tree with n points have one?)

Theorem Given any tree T and any point x in it, we can find a unique path, $p(x)$, whose last point is x. The length of the path $p(x)$ is the level (depth) of the point x.

Here is a more symbolic statement of the theorem:

$$\forall T : Tree. \forall x : Points(T).\ \exists p : Paths(T).$$
$$(last(p) = x \in Points(T)\ \&\ len(p) = lev(x))$$
$$\&\ \forall q : Paths(T). last(q) = last(p) \Rightarrow (q = p \in Paths(T)).$$

It is convenient to think of finite paths as lists, say $[s1, s2, \ldots, sn]$. The theorem calls for us to build a path where $sn = x$. It is easy to do this starting with x and working our way back to the origin using the predecessor function, i.e. building the list r defined as $[x, pred(x), pred(pred(x)), \ldots, s1]$. We notice that reversing r gives us the list we want, i.e. $p = reverse(r)$. We can either define r with an algorithm or define it implicitly by a proof. Here is the algorithm.

$$rpath(x) == if\ x = s1\ then\ [s1]\ else\ cons(x, rpath(pred(x))$$

where the operation $cons(x, L)$ constructs a new list whose head (first element) is the point x and whose tail is L. The cons operator is called the list constructor; so cons stands for constructor. We write [ ] for the empty list, also called nil in Lisp. That is nil = [ ]. Notice that $cons(x, nil) = [x]$. We call the lists built with elements from S, $List(S)$ or $S\ List$.

Now let $path(x) == reverse(rpath(x))$ for any x in $Points(T)$.

The proof that implicitly builds $path(x)$ proceeds by induction of the depth of the tree. We show that we can build the path for all points of depth 1 (base case) and that the length of the path is 1. That is trivial, the path is just $[s1]$. For the inductive proof, we assume that we can build path(y) for all points y whose depth is n, i.e. $lev(y) = n$, and that the length of $path(y)$ is n. Now we must show that we can build the path for all points x of depth $n + 1$ and that the length will be $n + 1$. Given point x of depth $n + 1$, by the induction hypothesis, we can build a path for $pred(x)$ since by properties $C2$ and $C3$, $lev(x) = lev(pred(x)) + 1$. Now the path we want is $[s1, \ldots, pred(x), x]$. We could make this more formal by proving a lemma that given any path p such as $path(pred(x))$, we can form a new path which is longer by one by "adding one element to the end" and defining a function, say $adjoin - to - end(p, z)$ for p a list (or path) and z a point.

Thus $path(x) = adjoin - to - end(path(pred(x)), x)$, which is exactly $[s1, \ldots, pred(x), x]$.

Exercise: Define the function adjoin-to-end either explicitly by recursive definition or by a proof that such a function exists.