[1] We're looking at the most basic data structures. Bunches are [2] unpackaged and unindexed. Sets are [3] packaged but not indexed. [4] Strings are indexed but not packaged. [5] And lists are both packaged and indexed. We've done bunches and sets. Now we'll do [6] strings. A string is an indexed sequence. [7] nil is the empty string. And [8] here is a 1 item string. In a bunch or set we talk about its elements. In a string or list, we talk about its items. There's no difference between a 1 item string and the item. Any number, any character, any binary value, any set, and any list is a 1 item string. Now [9] here's a 4 item string. Semicolon is the join operator; it joins strings together. [10] Here's the string length operator, saying the length of this string is 4. You could [11] measure its length by placing it on a string measuring ruler. The numbers on the ruler are called [12] indexes. Notice that the indexes come between the items. That way, when you're at [13] index 2, there's no confusion about which items have been processed, and which ones still remain to be processed. If index 2 were directly under an item, it wouldn't be clear whether that item has been processed or not. That's why the writing cursor in your word processor or text editor goes between the letters, instead of flashing a letter. And the [14] length wouldn't look right either. So if you're ever drawing a picture, put the indexes between the items. [15] Items are indexed from 0. So in this string, item 2 is the 7. That way, [16] when you're at index n, the number of items that have been processed is n, and the next item to be processed is item n. There aren't any plus 1s or minus 1s to think about. It's always n. The [17] next example just shows that you can index a string with a string of indexes, and get a string of results. [18] Now I want to spend a little time talking about indexing from 0. Zero was invented after the positive numbers, the negative numbers, the fractions, and even after irrational numbers. Old Euclid knew about irrational numbers 2 thousand 5 hundred years ago. But 0 started being used only about 1 thousand years ago, and is still not completely accepted as a number.

Actually, there are many people who have a very poor understanding of numbers. [19] There is a wonderful book by John Allen Paulos titled *Innumeracy: Mathematical Illiteracy and its Consequences*, and its successor, titled *Beyond Numeracy*, that tell just how badly people understand numbers. They're not required reading for this course, but if you have time, I recommend them. [20] I have seen this price tag on something that costs 10 cents. It says it costs a tenth of a cent. And I have seen a [21] gas price written like this. What's the second decimal point for? It's just wrong.

Now back to zero. Many people seem to dislike the number 0; some don't even accept it as a number. If someone says [22] “there are a number of things to discuss”, they don't mean there might be 0 things to discuss. They don't know what 0 means, and they don't know what to do with it. [23] On a tax form or accounting form, you sometimes see the directive “Subtract line A from line B; if there is no difference, write “nil”.”. On your [24] keyboard and telephone, 0 is placed after 9, which is mathematically silly.

[25] Here is a page from the 1991 Canadian census. Step 7 asks: how many persons who have a usual home somewhere else in Canada stayed here overnight between June 3 and June 4, 1991. Then there's a circle to tick off if there were none, and a box to fill in if there were some. So what's the point of the circle? If the right answer is 0, why not just fill in 0 in the box? In fact, when they get back the census forms, they do key in 0 for anyone who ticked off the circle. But the people who make up the form are smart. They know that if they didn't put the circle there, they would get a lot of phone calls from people who say they can't answer the question, because there were none.

[26] Here's a page from the front of the Toronto phone book from 1991. The column I'm interested in says TD at the top. It's the time difference between Toronto and other places. They put plus 6 to Cameroon, and minus 4 to the Cook Islands, so they know about positive and negative numbers. But to Cuba it says NA, and if you look up at the legend it says time difference not applicable. I guess they knew something was wrong because [27]

here's the 1993 phone book, and if you can find Haiti you see it has an equal sign. Everyplace else gets a number, but Haiti gets an equal sign. And that's they way it stayed until [28] 1997, when Bell Directories finally discovered the number zero, which is the time difference to Colombia. But they still felt the need to explain that one number in the legend; it says no time difference.

[29] When you measure, you must start at 0. If your measuring tape or ruler or scale does not start at 0, then it gives wrong answers. [30] And counting is measuring. If your odometer on your car didn't start at 0, then your car was used when you got it. After you're born, you are 0 years old for a whole year before you are 1 year old. Any other way of counting gives wrong answers. [31] For example, in music, they start counting intervals at 1. An octave is an interval of 8. What interval is 2 octaves? Care to guess? [32] Well, it's 15. Musical arithmetic doesn't work because they start from 1. [33] Year arithmetic doesn't work because the year 0 was left out; from a specific date, say July 1, in year X to the same date in year Y should be [34] Y–X years, and it would be if the year 0 had been included in the year numbering, but it's off by one if the interval crosses the boundary between BC and AD. The debate a few years ago about whether the new millennium starts in 2000 or in 2001 was about exactly that point.

[35] The Fortran language of 1955 had a loop construct whose body had to be executed at least once; I suppose it seemed senseless to have a loop whose body might be executed 0 times. Today, we are quite used to counting as follows: [36] count gets 0; while there's another one, increase count by 1. We start the count at 0, and the body of the loop might be executed 0 times. Just imagine how many bugs were caused by starting at 1, or by requiring that there be at least one iteration of the loop. The error was corrected in [37] Algol in 1958, and in PL/I, and in Pascal, in part: iteration might be 0 times, but the data structure over which one is iterating, the array, had to have at least 1 element. In Pascal that meant there was no empty string, and that set the algebra of data structures back about 50 years.

[38] For mathematics, and for computation, it is clear that we must count from 0. Thanks to the circuit builders, we are used to memory addresses starting with 0. And from the C and Java programming languages, we are used to indexing from 0. But for many people, it still seems wrong to start from 0, and right to start from 1. That's because we were taught to count from 1 when we were very young. We tend to believe whatever we are taught when we are very young as if our lives depended on it, because they often do. Evolution has thus favored uncritical infants. As a result, we hold a lot of critically unexamined beliefs. Whenever someone challenges something we learned when we were very young, we may feel threatened, as if our foundation is crumbling. Just the existence of people without those beliefs is a threat to those beliefs. People go to war (they are at war right now, killing each other) to defend their cherished but insupportable beliefs. The feeling of rightness when we count 1, 2, 3, ... as we were taught, and that any other way is wrong, is just that same defense. In this course, anything under my control will be numbered from 0. Anything not under my control will probably be numbered from 1.

[39] One of the problems is the association of the word “first” with the number 1. In English, there is no connection between them; “first” means “preceding all others in time, order, or importance”. The first channel on my TV is Channel 2. The first house on my street is numbered 35. The first hour of the day is numbered 12 on most clocks, and 0 on the 24-hour clocks. The first year of my life is numbered 0. The words “first” and “second” have no more claim to any particular number than does the word [40] “last”. — The word [41] “second” means “following the first”. But the association between first and 1 is so common that many people write [42] one s t for “first” as though 1 were "fir". Similarly, they write two n d for “second” as though 2 were “seco”. It makes just as much sense, maybe even more, to write [43] “first” as zero s t, that's “first”, and “second” as one n d. Well, let's get

[44] rid of those quickly. [45] The words “third”, “fourth”, and so on, are attached to numbers, but which ones? Your third year of life is age [46] 2. If you start with the first annual picnic in year 2012, in which year do you have your [47] 10 year celebration? Is that the 10th annual picnic? [48] The Merriam-Webster dictionary says [49] “the eleventh hour” means “the latest possible time”, but is the eleventh hour of the day the last hour (from 11 o'clock to midnight)? Is it even the last hour of the last half of the day? Or is it the hour from [50] 10 o'clock to 11 o'clock, in which case it isn't the latest possible time? My advice is: avoid the words [51] “third”, “fourth”, and so on. They are too error prone. Never say [52] “the fifteenth item”; say “item 15” if that's the one you mean, or “item 14” if that's the one you mean. [53] The first item in a string is item 0. It is not the “zeroth” item; that's what people say when they are unable to distinguish between “first” and 1. [54] If you mean the item that comes first, say the first item, or item 0.

Well, that was a long rant. [55] Let's get back to strings. There is an [56] ordering on strings, called lexicographic order. It's the same order used in dictionaries. The first item where two strings differ determines the order. Here it's the 6 coming before the 7. [57] If one string runs out before there's a point of difference, the shorter string comes first. [58] Here's a notation like we had for bunches, but there's a semicolon here instead of a comma. It means the string of integers starting at x and going up to but not including y. So the [59] length of x to y is y minus x, with no plus 1 or minus 1 to worry about. And if you [60] join x to y and y to z, you get x to z. There's no gap and no overlap.

There's a [61] special notation for text, which means a string of characters. You just put the characters between a pair of left and right double-quotes. And if you want a double-quote mark in the text, as in this example, you have to put it twice. [62] Of course, you could also write it out long, like this. Since a text is a string, [63] we can use string operators, like indexing. And if we use a string of indexes, we get a subtext.

Now [64] here's a string of 3 items. Item 0 is the bunch nat; item 1 is 1; and item 2 is the bunch 0 to 10. Joining strings, the semicolon, distributes over bunch union, just like addition and subtraction do. So a string of bunches is a bunch of strings. This one is all those strings whose item 0 is in nat, whose item 1 is 1, and whose item 2 is in 0 to 10. And [65] here is just one such string. The inclusion has to hold item by item. [66] There's a repetition operator. This is a repetition of 3 times the string 4 5. And the [67] one operand version, which is often called the Kleene star after the great logician Stephen Kleene, means any number of repetitions of the string. And finally, [68] there's an operator to make a new string that's just like an old one except that one item is replaced. So that's it for strings.

Take a deep breath, or stretch a bit, and [69] we'll go straight on to lists. [70] A list is a string in a package, just like a set is a bunch in a package. Square brackets are the list formation operator. This is a list of 3 items. [71] Here's another list of 3 items. Item 0 is nat; item 1 is 1; and item 2 is 0 to 10. List formation distributes over bunch union, so a list of bunches is a bunch of lists. And [72] the list 0 1 2 is one of those lists. And all those lists are [73] included in the bigger bunch consisting of all lists of 3 naturals. And they are [74] included in the even bigger bunch of all lists of naturals of any length.

It may sound strange to say a list of bunches is a bunch of lists. The math isn't strange; that's just a distribution law. What's strange is the English. A list sounds like one thing, and a bunch sounds like it could be many thngs. But a list with an item that's not an element is not an elementary bunch, even though it is a singular noun in English. To help you see the pattern, let me show you other distribution laws. [75] The negation of a bunch is the same as a bunch of negations. [76] The product of sums is the same as a sum of products. [77] A conjunction of disjunctions is the same as a disjunction of conjunctions. And [78] a list of bunches is the same as a bunch of lists. So that's the pattern. [79]

[80] Next is the content operator, which is the inverse of list formation, and produces the string which the list contains. [81] Then the list length operator. Do you see why it has to

be a different operator from string length? — What would we get if we applied the string length operator to that list? — We'd get 1, because a list is a single item. [82] Then indexing, which is of course from 0. [83] The domain of a list is the bunch of its indexes. [84] Then list composition. That means indexing a list with a list of indexes, and we get a list of results. [85] The list join operator is a pair of semicolons. In all the programming languages I know, there's no agreement on the join operator symbol, so I hope this one is ok with you. Lists are ordered [86] lexicographically, just like strings. [87] And the last list operator is a bit complicated because it has 3 operands. Modification takes an index, and an item, and a list, and produces a new list that's a lot like the old one except that at the given index there's the given item. Maybe we could say it like this: 2 maps to 22 and otherwise it's 10 to 15.

[88] I want to talk about modification for another moment, just to try to prevent an error that some people might make. Suppose L is the list 10 to 15, and we want to evaluate 2 maps to L 3 and 3 maps to L 2 and otherwise L. Well, first of all let's write out [89] list L. Now we [90] start with L, [91], there, and we [92] modify item 3 to be what item 2 is. [93] there. And now we [94] modify item 2 to be what item 3 is, well it already is the same, so we're done. And that's the [95] wrong answer. Do you see what went wrong? Let's [96] start again, and do it right. Starting [97] with L was correct, [98] there, and we [99] modify item 3 to be what L 2 is. [100] there. And now we [101] modify item 2 to be L 3. [102] There. The point is that all 3 of the Ls in that expression refer to L, the original list, not to some list that's part way through modification. I hope that clears up any possible misunderstanding about that.

[103] Indexing distributes over bunch union, no matter whether you are indexing a string or a list. If you use a bunch of indexes, you get a bunch of items. [104] Likewise if you use a set of indexes, you get a set of items. If you use a string of indexes, you get a string of items. And if you use a list of indexes, you get a list of items. Putting them all together, if you index with any complicated structure, you get that same structure of items. [105] Here's an example of a string being indexed with a structure, resulting in that same structure of string items. And if you index a list with that structure, you get that same structure of list items. There was a programming language called APL long ago with that feature, and I think currently the language J has it. It's really handy.

[106] The list examples so far have been one dimensional. But a two dimensional list is nothing new. It's just a list whose items are lists. A two dimensional list used to be called an array, but now programmers use the word array for any number of dimensions. This particular array [107] is included in the 3 by 4 arrays of naturals. If we [108] index it with just a 1, we get row 1. That's a list, so we can [109] index it, with a 2, say, and we get its item 2, which is the array item from row 1 and column 2. The main point I'm making now is: [110] if you want an item, don't use a comma, and don't use brackets. We're using comma consistently for bunch union, so the first one is a [111] bunch of 2 rows, [112] and that's not what you wanted. And the [113] second one is also not [114] what you wanted. What you wanted is [115] this.

[talking head] This may seem like a lot of notation to you. We've covered most of the notations of the whole course. We just have functions to go. This may be the first time you've seen a comprehensive and consistent set of theories. Usually you take binary theory in one course, set theory in another, and list theory in yet another, and they don't fit together because of a clash of notations, and you don't have all the axioms, so things aren't perfectly well defined, and you can't make any formal proofs. I guess that's ok if you only prove things informally, or don't prove them at all, and it doesn't really matter if you get it wrong. But if you have to get it right, or you want a theorem prover to help you, or if you want to build a theorem prover to help other people, you would have to have a comprehensive and consistent set of notations. So now you have. Mathematicians are usually not so picky about their notations. They can often get away with saying: oh come on, you knew what I meant.

But there's no use saying that to a compiler, or to a prover. You have to use exactly the notations they use. In this course, think of the marker who marks your tests as a compiler or prover. Use exactly the right notations.