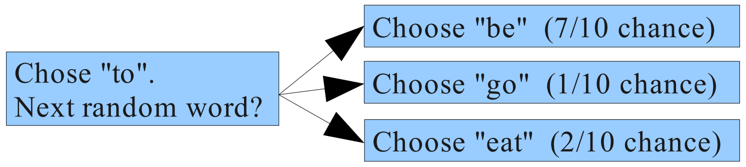
CSE 341, Autumn 2010
Assignment #4 - ML: N-grams (100 points)
Due: Friday, October 29, 2010, 8:30 AM

Perhaps you have heard of the "Infinite Monkey Theorem," which states that a monkey sitting at a typewriter pressing random keys for an infinite amount of time will eventually produce the complete works of William Shakespeare. Of course, the odds of the monkey typing exactly the right character each time become very low the more keys he presses. Suppose we have a slightly different goal: Could the monkey typing randomly on his typewriter produce a *new work* that "sounded like" Shakespeare's works, written in a similar style, with similar vocabulary, wording, punctuation, etc.?

Just like the original problem, it would take the monkey (us) almost forever to produce such a work. But let's consider a simplified and constrained version of the problem: What if we were choosing *words* at random, instead of letters? At least then the randomly generated text would consist of legal English words, though perhaps in a nonsensical order. Now suppose that rather than each word having an equal probability of being our choice, we *weighted* the probability based on how often that word appeared in Shakespeare's works? This would tend to produce text with word counts closer to Shakespeare's works. But the sentence structure and overall semantics would likely be nonsense.

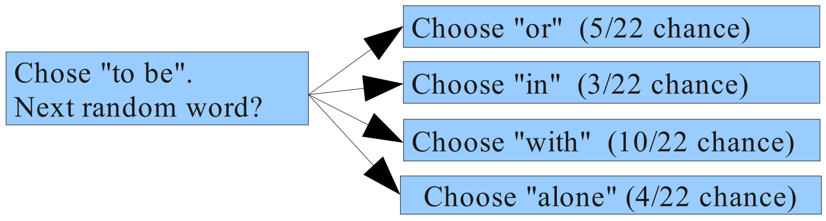
We can instead look at *chains of two words* in a row in Shakespeare's works; for example, perhaps in Shakespeare's works, the word "to" occurs 10 times total and is followed 7 of those times by "be", 1 time by "go", and 2 times by "eat". We can use those ratios when choosing our random next word. If the last word we chose is "to", we would randomly choose "be" as the next word 7/10 of the time, "go" 1/10 of the time, and "eat" 2/10 of the time, as shown below. We would never choose any other word to follow "to".


Chose "to".
Next random word?
Choose "be" (7/10 chance)
Choose "go" (1/10 chance)
Choose "eat" (2/10 chance)

We call a chain of two words like this, such as "to be" vs. "to go", a *2-gram* from Shakespeare's work. If you compute a large collection of all words along with their possible following words with relative weights, you can use this to guide the selection of a second word given the first word (and then choose the third word based on the second word, and the fourth word based on the third word, and so on). The following is a random sentence generated from 2-grams of *Hamlet*:

Go, get you have seen, and now he makes as itself? *(2-gram)*

A 2-gram sentence isn't great, but let's look at chains of 3 words (*3-grams*). If we have chosen the words "to be", what next word should follow? If we had a collection of all sequences of 3 words-in-a-row with their probabilities, we could make a weighted random choice. Suppose Shakespeare uses "to be" 22 times and follows them with "or" 5 times, "in" 3 times, "with" 10 times, and "alone" 4 times. We could use these weights to randomly choose the next word:


Chose "to be".
Next random word?
Choose "or" (5/22 chance)
Choose "in" (3/22 chance)
Choose "with" (10/22 chance)
Choose "alone" (4/22 chance)

So now the algorithm to generate a random chain would be to pick the third word based on the first two, then pick the fourth word based on the second+third, and pick the fifth based on the third+fourth, and so on. Here is a random sentence built from 3-grams of *Hamlet*. Notice that it is a bit less nonsensical than the 2-gram sentence.

One woe doth tread upon another's heel, so fast they follow. *(3-gram)*

You can extend this idea as many levels as you like. We call the generalization of these levels of word sequences an *n-gram*. The higher a level you use, the more similar the new random text will be to the original data source. Here is a random sentence generated from 5-grams of *Hamlet*, which is starting to sound a lot like the original:

I cannot live to hear the news from England, But I do prophesy th' election lights on Fortinbras. *(5-gram)*

Each particular piece of text randomly generated in this way is also called a *Markov chain*. Markov chains are very useful in computer science and elsewhere, such as artificial intelligence, machine learning, economics, and statistics.

Implementation Details:

The next several pages describe our overall strategy for implementing our solution to the n-gram problem. We will once again use the utility functions defined in `utility.sml`. (This file is unchanged since the last assignment.) You will also need a file `ngram.sml` that has supporting functions for this assignment. Put your code in a file called **`hw4.sml`**.

Since this is a large and complex problem, we'll break it apart into several major sub-tasks that combine to solve it. *NOTE:* `ngram.sml` has several debug functions, such as to print a tree of n-grams or a single n-gram. Check them out!

Part A - Processing the Input File:

You will write functions to analyze a text file and convert it into a series of n-grams, then use those n-grams to generate random text similar to the original. We will provide you with code to read in the contents of an input file and turn them into a large list of strings, where each string represents one word of the file. You will process this list to produce a collection of n-grams. The idea is to count the occurrences of word sequences of length *n* in a text.

Looking at actual 2-grams of Shakespeare's *Hamlet*, the words "To be" occur 6 times. So we would say that a 2-gram of this text is (["To", "be"], 6). We will include commas and some other punctuation as part of words; there happens to be another 2-gram of (["To", "be,"], 1). Note the comma after the word "be,". The number 1 means that the input file contains only one occurrence of that pair of tokens in a row.

It would seem natural to store an n-gram as a list of *n* words along with an integer count. But that is not an ideal structure for generating random text. To generate n-grams (word chains of length *n*), what you want instead is a way to answer the question: Given the previous (*n* - 1) words, what should the next word (the *n*th word in the chain) be? So what we'll generate instead are tuples that contain a list of (*n* - 1) words as well as the *n*th word by itself.

In the remainder of this specification, the term *"leading words"* will refer to a set of (*n* – 1) words from the original document and the term *"completion word"* will refer to a possible *n*th word to follow those (*n* – 1) words. Each combination of a set of leading words with a completion word comprises a single n-gram.

---

1. `groupWords` `val groupWords = fn : string list * int -> (string list * string) list`

Define a function `groupWords` that takes a list of words (strings) and an integer *n* and that produces a list of tuples where each tuple stores the pair: (*n* - 1 leading words, *n*th completion word).

The function should use the first *n* words to form a tuple, then the *n* words starting with the second word, then the *n* words starting with the third word, and so on until it finds there aren't *n* words left to form a tuple. For example, given words:

```
val words = ["Twas", "brillig", "and", "the", "slithy", "toves"];
```

The call of `groupWords(words, 3)` should produce:

```
[(["Twas", "brillig"], "and"),
 (["brillig", "and"], "the"),
 (["and", "the"], "slithy"),
 (["the", "slithy"], "toves")].
```

And on the same list of words, the call of `groupWords(words, 4)` should produce:

```
[(["Twas", "brillig", "and"], "the"),
 (["brillig", "and", "the"], "slithy"),
 (["and", "the", "slithy"], "toves")].
```

Notice that if the list is of length *m*, then the function returns (*m* – *n* + 1) tuples. The value of parameter *n* should be 2 or greater; if not, raise an exception of type `OutOfRange` (you should define this exception type in your program). If the list does not have at least *n* words, return an empty list.

Hint: If your code is correct but your function has the wrong type, try explicitly specifying the type of its parameter(s). You may use anything you like from ML's standard libraries (such as the `List` library) to help you.

Part B - Grouping Words into N-Grams:

After grouping the input file into *n*-word tuples, we must aggregate together all information about each unique group of (*n* - 1) words. For example, if we're making 3-grams and the 2 leading words ["to", "be"] appear several times in the text, we want all information about those leading words in a single place so we can use it to make random completion words later.

To do this, we will manipulate instances of a data type called `ngram`, provided in `ngram.sml`. Each `ngram` represents a group of (*n* - 1) leading words and all *n*th words that follow them in the original file. An `ngram` has three components:

- a string list storing the (*n* - 1) **leading words**;
- an integer **total count** for how many times this set of leading words occurs in the original text; and
- a list of **(completion word * count) tuples** for the frequencies of all words that ever follow these leading words.

```
datatype ngram = Ngram of string list * int * (string * int) list;
```

For example, if we're producing 3-grams from *Hamlet* and the file's `groupWords` result includes the following tuples:

```
[..., (["do", "you"], "look?"), ..., (["do", "you"], "call"), ..., (["do", "you"], "think"),
 ..., (["do", "you"], "think"), ..., (["do", "you"], "go"), ..., (["do", "you"], "speak"),
 ..., (["do", "you"], "mark"), ..., (["do", "you"], "read"), ..., (["do", "you"], "think"), ...]
```

Then the `ngram` instance we'll produce for leading words "do you" will look like this (9 is the total count of occurrences):

```
Ngram(["do","you"],9,
  [("think",3),("speak",1),("look?",1),("go",1),("call",1),("mark",1),("read,",1)]) : ngram
```

For this Part B, we'll write a few useful functions to manipulate an individual `ngram` instance for one group of leading words. In the next part, we'll think about how to store a large collection of n-grams to cover the entire file.

2. `createNgram`   `val createNgram = fn : string list * string -> ngram`

Define a function `createNgram` that takes string list of leading words and a string representing a completion word and turns it into an initial `ngram` instance storing that data with a total count of 1. This is a bit like a constructor in Java.

For example, the call of:

```
createNgram(["to", "be"], "or")
```

should produce:

```
Ngram(["to","be"],1,[("or",1)]) : ngram
```

3. `addToNgram`   `val addToNgram = fn : ngram * string -> ngram`

Define a function `addToNgram` that takes an existing n-gram and a completion word (string) as arguments and that produces a new n-gram that includes the given word in that n-gram's list of completions.

If the given word already appears in the n-gram's list of completions, then the new n-gram should increase the frequency of that word by 1. If the word does not appear in the original n-gram, it should be added to the list with a frequency of 1.

The following sequence of calls shows the effects of calling this method on a given n-gram instance:

```
val ng  = Ngram(["to","be"],1,[("or",1)]);
val ng2 = addToNgram(ng, "or");       (*  Ngram(["to","be"],2,[("or",2)])            *)
val ng3 = addToNgram(ng2, "with");    (*  Ngram(["to","be"],3,[("or",2),("with",1)])  *)
```

4. `randomCompletion`   `val randomWord = fn : ngram -> string`

Define a function `randomCompletion` that takes an n-gram parameter and that chooses randomly from its list of completion words, using the existing statistical frequency of the various completion words to weight the random selection.

For example, if we have the following n-gram:

```
val ng = Ngram(["to","be"],3,[("or",2),("with",1)]);
```

Then the call of `randomCompletion(ng)` will randomly produce `"or"` 2/3 of the time and `"with"` 1/3 of the time. You may want to call the function `randomInt()` that is part of `ngram.sml` and that returns a random `int` value.

Part C - Building an N-gram Tree:

Now we can group words together into n-gram instances. But we need a large data structure to store all of the n-grams from the input file. Iterating over the tuples from `groupWords` and turning them into an `ngram list` might seem like a good approach. But think about the algorithm we need to implement:

```
for each ([leading words], completion word) tuple in the input file:
    if we have already created an n-gram for these leading words,
        add this completion word into that n-gram.
    else, create a new n-gram for these leading words, and add it to our collection.
```

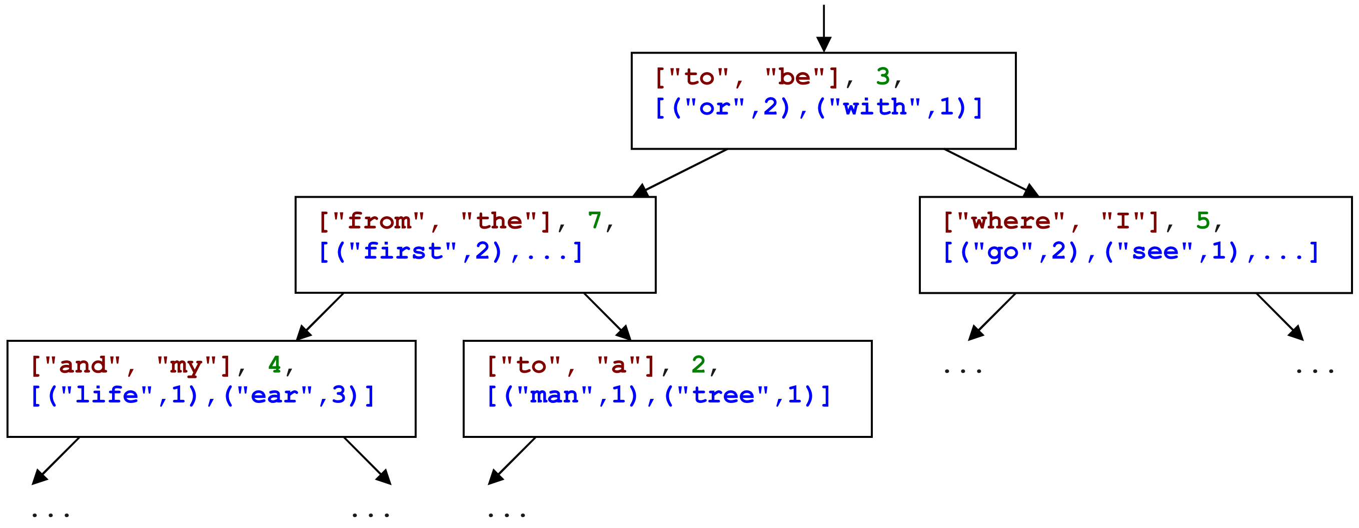
The key question in the above algorithm is: Have we already created an n-gram for a given list of leading words? A list is a poor data structure for answering that question, because we would have to do a sequential search through the entire list of n-grams, looking for one that had those leading words. Having to do this for every tuple in the original input would make the process impractically slow.

Instead, we will use a *map* as our data structure. Recall that a map is a collection that connects pairs, where each pair contains a *key* and an associated *value*. In our case, a key is a list of (n - 1) leading words, and a value is an `ngram` instance with information about those leading words and their nth word completions. If we had such a map, we could use it to quickly answer the question of, "Have we already created an n-gram for these leading words?"

You may remember from past coursework that a map can be implemented efficiently with a binary search tree (like Java's `TreeMap`) or a hash table (like Java's `HashMap`). We will implement ours as a *binary search tree* of n-grams. We won't use an existing library for our binary search tree; we will implement the tree ourselves as a collection of nodes.

Each node of our binary tree will store one `ngram` instance along with links to left and right subtrees. The data will be stored in typical binary search tree order, sorted by alphabetical order of the (n - 1) leading words of each `ngram`. That is, leading word sequences that occur earlier in the alphabet will be further to the left in the tree, and leading word sequences that occur later in the alphabet will be further to the right. For example, the leading words "to be" occur to the right than "and my", "from the", and "to a", and they occur further to the left than "to your", "where I", and so on.

The following diagram is a rough depiction of such a possible tree, made from fictitious input data:



If we have a tree such as the one above, our algorithm for processing the input data becomes:

```
for each ([leading words], completion word) tuple in the input file:
    if our tree already has an n-gram for these leading words,
        add this completion word into that n-gram.
    else, add a new n-gram into the tree for these leading words.
```

The file `ngram.sml` has the following definition for a data type we'll use in the tree:

```
datatype NgramTree = Empty | NgramNode of ngram * NgramTree * NgramTree;
```

Each black box in the above diagram can be thought of as an instance of type `NgramTree`.

(The functions to implement for this part are described on the next page.)

5. stringListCompare `val stringListCompare = fn : string list * string list -> order`

Define a function `stringListCompare` that takes two string lists as arguments and that indicates how the lists compare by producing a value of type `order`. You'll use this function to know which way to go when you traverse an n-gram tree.

We haven't discussed type `order`, but it is very simple. (It is also described on page 325 of the textbook.) It is a built-in type that is part of ML's Standard Libraries, that is used to represent < = > ordering. It has the following definition:

```
datatype order = LESS | EQUAL | GREATER;
```

Your function should compare the lists and return `LESS` if the first list comes alphabetically earlier than the second; `EQUAL` if the two lists are exactly the same; and `GREATER` if the first list comes alphabetically later than the second.

Compare words in pairs. Compare the first list's first word to the second list's first word; then the lists' second words, and so on. You're looking for the first pair that are not the same. If a pair differs, produce `LESS` if first list's word is less than the second list's or `GREATER` if the word from the first list is greater than the word from the second list. For example:

`stringListCompare(["hi","`**`how`**`","ru"],["hi","`**`yo`**`","man"])` produces `LESS`
`stringListCompare(["hi","how","ru"],["hi","how","ru"])` produces `EQUAL`
`stringListCompare(["hi","how","`**`ru`**`"],["hi","how","`**`me`**`"])` produces `GREATER`

If the lists are not the same length, compare them as usual; but if they are entirely equal up until one list runs out of elements, the one with elements remaining is considered to come alphabetically later. For example:

`stringListCompare(["hi","how","`**`ru`**`"],["hi","how"])` produces `GREATER`
`stringListCompare(["`**`hi`**`"],["`**`bye`**`","now"])` produces `GREATER`
`stringListCompare(["`**`hi`**`"],["`**`you`**`","go","boy"])` produces `LESS`

6. addToTree `val addToTree = fn : NgramTree * string list * string -> NgramTree`

Define a function `addToTree` that takes an n-gram tree, a list of leading words (strings), and a completion word (string) and that produces the tree obtained by inserting the leading words/completion word combination into the tree.

If the set of leading words is already in the tree, the completion word should be added to that n-gram. If not, a new n-gram should be added to the tree with a frequency of 1. The tree should be ordered based on how the strings compare with each other. If the n-gram tree is empty, the result is a new tree with the given data as its sole element (its root).

For example, if a variable called `tree` stores the tree drawn on the previous page and we make the call of:

```
addToTree(tree, ["from", "the"], "first")
```

Then the result should be a tree with the same structure, except that the node left of the root has been updated to have an overall count of 8 and a "first" completion word count of 3. If we instead made the call of:

```
addToTree(tree, ["to", "all"], "the")
```

Then the result should be a tree with similar structure, except that there should be a new node to the left-right-right of the root that stores a new n-gram: `Ngram(["to", "all"], 1, [("the", 1)])` .

Recall that you can use the `as` keyword to describe a parameter and its parts, as in our lecture 10 `IntTree add` code. You may also want to use the function `getWords` from `ngrams.sml` gets the list of leading words from an `ngram`.

7. addAllToTree `val addAllToTree = fn : (string list * string) list -> NgramTree`

Define a function `addAllToTree` that takes a list of tuples of the form (leading words, completion word), and that returns an n-gram tree built by inserting each tuple's data into an initially empty n-gram tree (they can be added in any order). The idea is that this method will be called on the input groups read from the input file to turn all of it into one large tree.

Throughout this assignment we will assume that we are working with **valid data**, legal n-grams, and legal n-gram trees. In particular, we will assume that we are working with "≥ 2"-grams, that the overall frequency in an n-gram is equal to the sum of the individual frequencies of its completion words, that no completion words have a frequency of 0 or less, that the list of completion words is nonempty, that the n-gram tree is a proper binary search tree based on the leading word lists, and that the n-gram tree is nonempty. We will also assume that data passed to our functions are consistent in terms of n-gram length. For example, we will assume that we won't encounter both a 2-gram and a 3-gram in the same tree.

Part D - Generating Random Text:

Now that we can build n-gram trees, let's use them to produce random text. There are two overall problems to solve:

- Producing text relies on looking at the last (n - 1) words. But how do we start, when there are no previous words?
- Once we have already generated some number of preceding words, how do we generate the next word?

8. **randomStart** `val randomStart = fn : NgramTree -> string list`

Define a function `randomStart` that takes an n-gram tree parameter and randomly picks a list of (*n* - 1) leading words where the first word begins with a capital letter. We'll use this function to begin the process of making our random text.

We look for the capital letter to make sure we begin the document with a word that previously began a sentence, because we don't want our random document to start in mid-sentence. You may assume that the given tree contains at least one node whose first leading word begins with a capital letter.

Only some n-grams will begin with capitalized words. Your function must pick randomly among them with equal probability. You will need a way to walk your entire tree looking for all such n-gram nodes. Therefore this function is allowed to run in O(*n*) time for a tree with *n* nodes. You can use the function `isSentenceStart` from `ngram.sml` to help you; it accepts a string as a parameter and produces `true` if that string begins with an capital letter.

9. **lookup** `val lookup = fn : NgramTree * string list -> ngram option`

Define a function `lookup` that takes an n-gram tree and a list of leading words (strings) as parameters and produces an n-gram `option` that is either the n-gram associated with that set of leading words in the given tree, or `NONE` if not found.

For example, if a variable called `tree` stores the tree drawn on the preceding page, the call of:

```
lookup(tree, ["from", "the"])
```

would produce a result such as:

```
SOME Ngram(["from","the"],7,[("first",2),...]) : ngram option
```

On the same tree, the call of `lookup(tree, ["to", "all"])` would produce `NONE`.

Later functions you'll write might call `lookup` as part of their behavior. Recall that you can use the `isSome` and `valOf` functions to examine an `option` to see whether it is `SOME` value or `NONE`, or to extract the value from a `SOME`.

10. **buildTree** `val buildTree = fn : string * int -> NgramTree`

Define a function `buildTree` that takes file name (a string) and integer *n* as arguments and returns an n-gram tree constructed using the words from the given input file with n-grams of length *n* (i.e., each having (n – 1) leading words). For example, the call `buildTree("hamlet.txt", 3)` will produce a tree of 3-grams from *Hamlet*. For a large text file like Hamlet, this method may take several (5-10) seconds to run.

You can call the function `read` included in `ngram.sml` to convert the file into a list of tokens. You may assume that the given file exists and is readable. If *n* is less than 2, you should raise an exception of type `OutOfRange`.

11. **randomDocument** `val randomDocument = fn : NgramTree * int -> string list`

Define a function `randomDocument` that takes an n-gram tree and an integer *count* as parameters and that produces a list of approximately *count* words that are randomly generated using the given tree. This is it! The grand finale.

Below is a rough pseudo-code description of how the list should be constructed:

- Choose (*n* - 1) random starting words, beginning with a capitalized word. Include these words in your result.
- While not done:
  - Choose a random completion word for the current "window" of (*n* - 1) most recently chosen leading words.
  - Include the new completion word in your result, and shift over your "window" by one word.

The list of words that you generate should end with a word that is the end of a sentence (a string ending with `.`, `?`, or `!`). You can use the function `isSentenceEnd` included in `ngram.sml` to check for this.

It's possible that you will generate exactly *count* words, but more often you must generate extra words so your document consists of complete sentences. Continue until both constraints are satisfied (at least *count* words; complete sentence).

For example, suppose that you were working with a tree of 3-grams generated from *Tom Sawyer* and you are asked to build a random document of 4 total words. The sequence of choices might go like this:

Action(s)	Current (*n*-1) "window"	Current result
choose a random start	`["But", "all"]`	`["But", "all"]`
choose new word; shift	`["all", "things"]`	`["But", "all", "things"]`
choose new word; shift	`["things", "are"]`	`["But", "all", "things", "are"]`
choose new word; shift	`["are", "gone."]`	`["But", "all", "things", "are", "gone."]`

In this case, we actually had to pick 5 words before we could stop, because it took that long to find a word that could end a sentence. Notice that we always have a two-word window, since we are working with 3-grams.

There is one special case you must handle. It will occasionally happen that the most recent (*n* - 1) leading words are not in the n-gram tree. This happens when you use the very last (*n* – 1) words that appeared in the original text; unless those exact (*n* - 1) words also appeared together earlier in the document, there is no *n*th word after them and therefore no n-gram node corresponding to them in the tree. If you tell your `randomDocument` function to generate a long enough document (such as 99999 words), you'll start to see this error often; you may get an `Option` exception.

When this case occurs, add all of those (*n* - 1) words to your result, and then if you are still not finished generating your document (if you have not yet produced *count* words), resume the overall algorithm with a new random start of (*n* - 1) words. We'll assume that the last word of the file can end a sentence even if it doesn't end with a `.`, `?`, or `!`.

Once you finish this function, you will probably want to construct a tree of n-grams for a particular text, then generate a random list of words and then call the `printList` function from `ngram.sml` to display the results, as in:

```
- val tree = buildTree("hamlet.txt", 3);
- val list = randomDocument(tree, 100);
- printList(list);
Horatio, and Marcellus. Fran. I think nothing, my lord. Ham. Arm'd, say
you? Both. Arm'd, my lord. Ham. I will be brief. Your noble son is mad. Mad ...
```

Debugging Tips:

This program is hard. It can be tough to figure out why you have a given bug or how to fix it. Here are some debug tips.

If you want the ML interpreter to display the content of complex structures at a deeper level of nesting, use a line such as:

```
Control.Print.printDepth := 10;
```

If you want to see a full **stack trace** when an exception occurs, put the following two lines at the top of your code:

```
CM.make "$smlnj-tdp/back-trace.cm";
SMLofNJ.Internals.TDP.mode := true;
```

The `ngram.sml` file has several helpful **debugging functions**. In particular, you can pass an `ngram` instance to the `ngramToString` function to see a nice printout of its contents, and you can pass an `NgramTree` instance to the `printTree` function to see its nodes printed sideways with proper indentation.

Make sure to test each function before you move on to the next function. Test each function with a very small and simple input first. For example, there is a provided input file called `tiny.txt` with a small number of words. If you're building a tree, try using that file first, because it is not impractical to print the entire tree and examine it to verify its structure.

Extra Credit (+1 point) - Custom Input File and Output:

For an additional +1 point of extra credit, create your own input file called `myinput.txt` containing a fairly large amount of text of your own choosing. You can use a file(s) that you found on the internet, and the content can be anything you like, so long as it is not extremely similar to the existing input files provided on the web site.

Also submit a file `myoutput.txt` that contains one or more randomly generated stories made from your `myinput.txt`.

Grading and Submission:

For reference, our solution is 63 "substantive" lines long according to the class Indenter page, excluding blank lines and comments. You don't need to match this number or even come close to it; it is just a rough guideline.

Your program should not produce any syntax errors or warning messages, such as "non-exhaustive match." You may lose points if you name your functions or other top-level values incorrectly, even if their behavior is correct.

Your code should work for both basic and advanced cases. Perform your own testing, and remember to test edge cases.

As always, if the solution to a previous problem is useful in helping you solve a later problem, you should call the earlier function from the later function. You can include any testing code you develop (although it should be labeled as such). Other than testing data/code, do not define any other global symbols not described in this document.

If you write inner helper functions, you should choose a suitable set of parameters for each one. Don't pass unnecessary parameters or parameters that are unmodified duplicates of existing bound parameters from the outer function.

You are expected to use good programming style, such as naming, indentation/spacing, and avoiding redundancy when possible. Avoid long lines of over 100 characters in length; if you have such a line, split it into multiple lines.

In general, favor the use of patterns rather than `if-then-else` to distinguish between cases. Favor the `::` operator to grow lists versus the `@` operator (though the `@` operator is sometimes necessary and is not forbidden entirely).

Place a descriptive comment heading at the top of your program along with a comment header on each function, including a description of the meaning of its parameters, its behavior/result, and any preconditions it assumes. If you declare a non-trivial or non-obvious inner helper function, also briefly comment the purpose of that helper in a similar fashion.

Efficiency on a fine level of detail is not crucial on this assignment. But in general, functions that process a list should run in O(*n*) time for a list of length *n*. Functions that process an n-gram tree should run in O(*height of tree*) time, except for the `randomStart` function that is allowed to examine the entire tree.

Redundancy, such as recomputing a value unnecessarily or unneeded recursive cases, should be avoided.

You may not use arrays, references, or vectors (we haven't covered these anyway), but otherwise you may use any ML constructs and functions from the ML basis library. You might find the libraries `List` and `Char` particularly helpful.