CS106X Midterm Examination

This is an open-book, open-note, closed-electronic-device exam. You have 90 minutes to complete it.

Good luck!

Section Leader: ______________________

Last Name: ______________________

First Name: ______________________

SUNet ID: ______________________**@stanford.edu**

I accept the letter and spirit of the honor code.

(signed) ____

		Score	Grader
1. Prefix to Postfix	[10]	______	______
2. Magic Numbers	[10]	______	______
3. Derivability	[10]	______	______
Total	**[30]**	______	______

Summary of Relevant Data Types

```
class string {
   bool empty() const;
   int size() const;
   int find(char ch) const;
   int find(char ch, int start) const;
   string substr(int start) const;
   string substr(int start, int length) const;
   char& operator[](int index);
   const char& operator[](int index) const;
};

class Vector {
   bool isEmpty() const;
   int size() const;
   void add(const Type& elem); // operator+= used similarly
   void insert(int pos, const Type& elem);
   void remove(int pos);
   Type& operator[](int pos);
   const Type& operator[](int pos) const;
};

class Stack {
   bool isEmpty() const;
   void push(const Type& elem);
   Type pop();
};

class Map {
   bool isEmpty() const;
   int size() const;
   void put(const Key& key, const Value& value);
   bool containsKey(const Key& key) const;
   Value get(const Key& key) const;
   Value& operator[](const Key& key);
   const Value& operator[](const Key& key) const;
};

class Set {
   bool isEmpty() const;
   int size() const;
   void add(const Type& elem); // operator+= also adds elements
   bool contains(const Type& elem) const;
};
```

Problem 1: Prefix to Postfix [10 points]

In lecture, we used a stack to evaluate postfix expressions of the form **`"45+9*73-24+**"`**. In the interest of simplicity, we constrained the operand domain to single digit integers so the entire expression could be easily captured in a string.

For this problem, you're to write a function that converts valid prefix expressions (e.g. **`"*-2+46-+259"`**) to the equivalent postfix expressions (e.g., **`"246+-25+9-*"`**). We'll again restrict our operand domain to single digit integers.

Your implementation must use a stack and conform to the following algorithm:

- Read the supplied prefix expression in reverse—that is, from back to front.
- Each time you encounter a digit character, push it to the stack.
- Each time you encounter a non-digit character, assume it's a valid operator, pop two items from the stack, and push the postfix equivalent of the relevant subexpression onto the stack as a string item.
- Repeat until the entire prefix expression has been processed and return the equivalent postfix expression as a string.

You should assume the supplied prefix expression is well-formed and that you needn't do any error checking whatsoever.

Use the rest of this page and the next to present your implementation:

```
static string prefixToPostfix(const string& prefix) {
```

Problem 1: Prefix to Postfix [continued]

Problem 2: Magic Numbers [10 points]

A pandigital number is a ten-digit number comprised of each of the ten digits in some order, e.g. 4106357289 is a pandigital number. As far as pandigital numbers go, 4106357289 has an interesting divisibility property. If we let **a** be the first digit, **b** be the second digit, **c** be the third digit, **d** is the fourth digit, etc., then we observe the following:

bcd = 106 is divisible by 2
cde = 063 is divisible by 3
def = 635 is divisible by 5
efg = 357 is divisible by 7
fgh = 572 is divisible by 11
ghi = 728 is divisible by 13
hij = 289 is divisible by 17

Numbers like this—where **bcd** is divisible by the smallest prime, **cde** is divisible by the second smallest prime, etc.—are called primal. When a number is both primal and pandigital, we say it's a **magic number**.

Looping through all 10 digits numbers to see which ones are incidentally magic is entirely too time consuming. However, it's possible to recursively discover all magic numbers by generating permutations of the string **`"0123456789"`** and relying on the **`stringToInteger`** function to convert strings to numbers. While constructing magic numbers, you can prune partial permutations when it's clear they're dead ends.

Use this and the next page for your implementation of **`generateMagicNumbers`**, which recursively discovers all magic numbers as described above and implants them in the **`numbers`** set. Your recursion **must** prune the search when it's clear there's no sense in continuing.

```
// kPrimes is a constant such that kPrimes[0] is 2, kPrimes[1] is 3, etc
static const Vector<int> kPrimes({2, 3, 5, 7, 11, 13, 17});
static void generateMagicNumbers(Set<int>& numbers) {
```

Problem 2: Magic Numbers [continued]

Problem 3: Derivability [10 points]

For this problem, you're to revisit the RSG problem from Assignment 2. Recall the simple context-free grammar presented in the handout:

```
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```

This particular grammar is capable of generating a small number of expansions, including **"The slugs die warily tonight."** and **"The big yellow flowers portend like big yellow flowers tonight."** Other grammars—**excuse.g** comes to mind—are capable of generating an unbounded number of expansions.

For this problem, you'll write a backtracking function called **cbg**, (short for **canBeGenerated**) to decide whether a specific expansion can be generated from the **"<start>"** symbol of a grammar. To illustrate, assume the grammar above has been compiled into a **Map<string, Vector<string>>** called **g**, where each key is a nonterminal (e.g. **"<object>"**), and each **Vector<string>** stores some key's productions (e.g. **["waves", "big yellow flowers", "slugs"]**). We'd expect **cbg** to behave like this:

```
cbg(g, "The slugs die warily tonight.") -> true
cbg(g, "The big yellow flowers portend like big yellow flowers tonight.") -> true
cbg(g, "The waves die like waves tonight.") -> false
```

Determining whether a particular expansion can be generated from **"<start>"** requires backtracking, since the expansion being matched may share a prefix with two or more productions at any point in the search. For instance, a variation on the above grammar might include the following:

```
<verb>
3
sigh like <object>
sigh <simile>
sigh like no other <object>
```

If you're trying to generate **"sigh like no other slug"**, it's possible the first is capable of generating it, or it's possible that the first isn't but the second is, etc.

Use the rest of this page and the next to implement the **cbg** function as described above. Your function needs to return **true** or **false**, but it doesn't need to return any information about what series of productions are involved in an expansion. You should assume the grammar is structured so that a properly implemented **cbg** won't get trapped in some infinite left recursion (e.g. **"<term>"** won't expand to **"<term>"** or anything else that might eventually begin with **"<term>"**).

You should implement **cbg** using a wrapper function to introduce the requirement that all expansions must be generated from **"<start>"**.

```
static bool cbg(const Map<string, Vector<string>>& g, const string& expansion) {
```

Problem 3: Derivability [continued]