COE428 Lab 5: Validating XML

Use of Stacks and Hash Tables

1. IMPORTANT: Two week lab

Note that you have two weeks to complete this lab. This lab must be submitted at least 48 hours before the beginning of your next lab period.

2. Prelab preparation

Before coming to the lab you should:

- Read the lab. Try to prepare any questions you may have about the lab.
- Refer to **Lab Guide**.
- Create your lab directory for lab5. (i.e. use **mkdir lab5** within your coe428 directory.)
- Change to your coe428/lab5 and unzip the lab5.zip file with the command:
  unzip /home/courses/coe428/lab5/lab5.zip

- Ensure that you have downloaded properly by entering the command: **`make`**

  No errors should be reported.

Requirements

The requirements to complete the lab are summarized below.

1. Complete Parts 1, 2 and 3.
2. Answer the Questions (see below) in the README file.

Theory

The eXtended Markup Language (XML) is a widely used format for describing structured data.

For this lab, we only examine a simplified form of XML: An XML document is a collection of matching start-tags and end-tags. A start-tag is a string of alphabetic characters preceded by a < and terminated with a >. For example <foo> is a start-tag of type "foo".

An end-tag is an alphabetic string preceded by </ and terminated by a >. For example </foo> is an end-tag of type "foo".

Well-formed XML requires that start-tags and end-tags match (i.e. have the same type.)

Examples		
XML	**Valid?**	**Explanation**
<a>	Yes	"a" tags balance
<a>	Yes	"a" outer tags and "b" inner tags balance
<a>	No	"a" end-tags does not match start-tag ("b")
<a><a>	Yes	all tags balance
<able><baker></Baker></able>	No	"Baker" end-tag does not match start-tag ("baker") (i.e. the tag names are *case-sensitive*.)

The Algorithm

The algorithm to determine if the start and end-tags balance uses a Stack data structure to keep track of previously read start-tags. The algorithm is:

1. Read the input until the beginning of a tag is detected. (i.e. tags begin with <: if the next character is a / (slash), then it is an end-tag; otherwise it is a start-tag).
2. Read the tag's identity. (e.g. both tags <x> and </x> have the same identity: 'x').
3. If the tag was a start-tag, push it onto the Stack.
4. Otherwise, it is an end-tag. In this case, pop the Stack (which contains a previously pushed start-tag identity) and verify that the popped identity is the same as the the end-tag just processed. **Note:** if the stack cannot be popped (because it is empty), the input is invalid; the algorithm should STOP.
5. If the identities do **not** match, the XML expression is invalid. STOP.
6. If they do match, then no error has been detected (yet!).
7. If there is still input that has not yet been processed, go back to the first step.

8. Otherwise (no more input) then the input is valid **unless** the Stack is not empty. Indicate whether the input is valid (Stack empty) or invalid and STOP.

Part 1: Validation of single-character tags

We begin with a simplified XML that restricts tag identifiers to single-character lower case letters. (i.e. there are only 26 valid tags).

Objective

Your program (called validateXML) reads *stdin* and determines if it is valid XML. If it is valid, it prints to *stdout* the message "Valid" and exits with an exit code of 0 (zero); otherwise, it prints "NOT Valid" and exits with an exit code of 1 (one).

Note

Nothing else should be printed to *stdout*. (For example, if the input is invalid, the program does not have to explain the problem it found.) However, additional information may be printed to *stderr*. Indeed, in the case of Stack underflow or overflow, a message should be printed to *stderr*.

C source code files

To implement the algorithm in C, you need a main() function that reads *stdin* and processes each character according to the algorithm. You also need to implement a Stack (including push(), pop() and isEmpty()) operations.

You must use separate C source code files for each of these. The files are:

part1Main.c

The main() function in this file reads *stdin* one character at a time and implements the algorithm. You are provided with a skeleton implementation of main() that handles the "read characters from *stdin* loop".

However, you have to code the actual algorithm. You will need the Stack for this; in particular, you will need to use the functions push() and pop().

These functions can be used (and their prototypes are included in the skeletal part1Main.c) but they must be implemented in the second source code file: intStack.c.

intStack.c

Again, you are provided with a skeletal version of this file. You need to implement the 3 functions (push, pop and isEmpty). The comments describing what these functions do must not be modified.

Part 2: Validation and counting of single-character tags

Objective

Your program (called countXML) works like the previous one but also keeps track of the number of times each start-tag is used. As before, the program should print "Valid" or "NOT Valid". In addition, if the input was valid, it should then print a table with a line for each start-tag encountered and a count of how often it occurred.

For example, given the input:

```
<x><a></a><b><a></a><y></y></b></x>
```

The output should be:

```
Valid

a 2

b 1

x 1

y 1
```

Counting the number of start-tags of each type should be done with a *direct-mapped table*.

C source code files

The countXMLsimpleTags program is built using two source code files: part2Main.c and intStack.c.

The implementation of the integer Stack in "intStack.c". Consequently, there is no need to modify this file for Part 2 of the lab.

The main() function for Part 2 incorporates all the features of the Part 1 version and adds functionality (counting tags). To get started, simply copy the file "part1Main.c" to "part2Main.c".

Part 3: Validation/counting of arbitrary tags

We now allow tag names of arbitrary length composed of upper- and lower-case alphabetic characters. Thus a tag like <Supercalifragilisticexpialidocious> is quite legal.

Although the overall architecture of the software to do this is the same as Part 2, the fact that the universe of keys is now unbounded (i.e. strings of arbitrary length vs. single lower-case characters) has consequences:

C programming (strings vs. characters)

Parts 1 and 2 of this lab did not have to deal with tag names that were more than 1 character long. Hence:

1. Reading the input one character at a time was sufficient.
2. The stack only had to have integer entries.
3. Comparison of an end-tag identifier with a popped identifier could be done with the == operator.

Alas, all of these simplifications do not apply when the tag ID is a string. In particular:

1. A different Stack implementation with entries that are strings (i.e. char pointers) is needed.
2. The string (char pointer) has to be allocated (and freed) dynamically.
3. Comparisons between strings to determine equality cannot be done with the == operator. (Use instead the "strcmp()" function.)

Direct mapping vs. Hash table

When the number of tag names is unlimited, direct-mapping of the name to an array index is out of the question. (Even if we limited tags to 9 characters, the universe of possibilities is more than 144 quadrillion—1.4 10^16!)

Since each tag has an associated count and we want to find the tag efficiently, a hash table must be used.

Objective

Your program ("countXML") should validate XML input and print "Valid" or "NOT Valid". When the input is valid, it should print a table of tags used and their frequency.

C source code files

You also need to implement a string Stack (including push(), pop() and isEmpty()) operations.

You must use separate C source code files for each of these. The files are:

part3Main.c

The main() function in this file reads *stdin* and implements the algorithm. You are provided with a skeleton implementation of main() in the file "part3Main.c"

stringStack.c

Again, you are provided with a skeletal version of this file. You need to implement the 3 functions (push, pop and isEmpty). The comments describing what these functions do must not be modified.

stringHashTable.c

Again, you are provided with a skeletal version of this file. You need to implement the add() function. The comments describing the function do must not be modified.

You may implement the hash table using any method (for example, chaining or open-addressing to resolve collisions). You can also use any hash function you wish (eg. division vs. multiplication). The size of the hash table is defined by a constant "HASH_TABLE_SIZE". The hash table itself must be an array of hash table entries. It must be a global variable called "hash_table".

Questions

Answer the following questions in your README file.

1. Which hash table collision resolution method did you use (eg. chaining or open addressing)? Explain your choice **briefly** (less than 25 words).
2. Which hash function (division or multiplication) did you use? How did you convert a string into a number?
3. Another legal XML tag not used in this lab is the "stand-alone" tag. This kind of tag combines both a start-tag and end-tag in one. It is identified with a '/' (slash) preceding the final >. (For example, the <foo/> is a stand-alone tag that is "self balancing".

   Describe **briefly** how you would modify Part 3 to allow this kind of tag.

Submit your lab

1. Go to your **coe428** directory
2. Zip your **lab5** directory by using the following command:
   zip -r lab5.zip lab5/
3. Submit the lab5.zip file using the following command:

   submit coe428 lab5 lab5.zip

by Ken Clowes, revised by Olivia Das