CSI 402 – Systems Programming

Programming Assignment IV

Date given: Nov. 10, 2017 **Due date:** Nov. 22, 2017
Total grade for this assignment: 100 points **Weightage:** 3%
Note: Programs that produce compiler/linker errors will receive a grade of zero.

A. Purpose. Assemblers (and other system–level programs) are expected to be very fast. Therefore, data structures that support efficient insertion and searching are needed to store assembler tables. In class, we have discussed two of the most important tables used by assemblers: (i) the `symbol table`, and (ii) the `machine opcode table`. The symbol and machine opcode tables are used during the first pass of a two–pass assembler for SIC/XE. Specifically, the symbol table is where the assembler keeps track of mappings between symbols (i.e., variables) and LC values (i.e., the address in the program where that variable is defined). A two–pass assembler must generate the symbol table in the first pass, and search the table in the second pass. The machine opcode table stores mappings between the instruction mnemonics (e.g. LDA) and their corresponding opcode (an integer). The assembler uses this table to convert the instructions into a binary format.

Your task in this assignment is to develop a program that will perform the above mentioned tasks on SIC/XE code as part of a full–edged SIC/XE two–pass assembler. Your program should store the symbol table in a `hash table`, and the machine opcode table in a `binary search tree`.

Hash Tables: A hash table is a common data structure that supports efficient insertion and retrieval of elements based on a key. In its simplest version, a hash table is implemented as an array. A hash table uses a `hash function` to compute an index into an array of buckets or slots, from which the desired value can be found. For example, suppose the hash function is $h(x)$, and the key for a object O is k. To insert O into the hash table, the index $y = h(k)$ would need to be computed, and the value O would be stored in the array at index y. Integer y is called the hash value of key k. Ideally, a hash function will assign each key to a unique bucket. Unfortunately, in practice, "perfect" hash functions are rare. Most hash functions are "imperfect", i.e., not one–to–one functions, which means that for two distinct keys k_1 and k_2, there is no guarantee that $h(k_1)$ and $h(k_2)$ will be distinct. In other words, the hash function generates the same index (i.e., hash collisions) for more than one key (in this case keys k_1 and k_2 have the same hash value). Such collisions must be accommodated in some way. When collisions occur, multiple objects cannot be stored at the same index in the array. Therefore, a `collision resolution strategy` is required.

For the purpose of this assignment, you will use a hash table to store symbol table entries (i.e., a struct containing a symbol and its corresponding LC value). The key will be the symbol itself. Additionally, you will use `chaining`, according to which, each element of the array is the head of a linked list. At index y in the array, the linked list contains all of the objects whose keys have hash value y. Thus, to insert an element O with key k into the hash table, the following steps need to be performed: (i) a new linked list node N needs to be constructed to store O, (ii) the hash value $y = h(k)$ must be computed, (iii) if $table[y]$ is null (i.e., the list at location y is empty), then simply

set $table[y] = N$, otherwise, insert N into the existing linked list stored in $table[y]$.

Binary Search Trees: A Binary search tree (BST) is a data structure that supports efficient lookup, insertion, retrieval, and removal of "items" based on a "key", and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name). A BST consists of nodes. Each node has a key, one or more data values, a left child, and a right child (the children are also nodes). At the "top" of the tree is a special node, called the root, which is not the child of any other node. Every node can be considered the root of a subtree that starts at that node and descends downward. Given a node N, the subtree rooted at N's left child is called the left subtree of N, and the subtree rooted at the right child of N is called the right subtree of N.

BSTs keep their keys in sorted order, so that lookup and other operations can use the principle of `binary search`. Specifically, given a node N with key k, the left subtree of N contains nodes whose keys precede k, while the right subtree of N contains nodes whose keys succeed k. The key can be any type which can be ordered (e.g., integers that can be compared or strings that can be lexicographically ordered). When searching for a key in a BST (or a place to insert a new key), the BST is traversed from root to leaf, making comparisons to keys stored in the nodes of the BST and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

B. Description. Your program will generate a machine opcode table, stored in a binary search tree, and a symbol table, stored in a hash table. The executable version of your program must be named `firstpass`. Your `makefile` must ensure this. The `firstpass` program must support the following usage:

```
firstpass instructionset programfile hashtablesize
```

- `instructionset` is the name of a text file containing a list of instructions. You can assume that each line of the file has the form: `instruction opcode format`, where `instruction` is a string, `opcode` is a decimal integer, and `format` is a decimal integer in the range $1 \geq format \leq 3$. The string is a mnemonic for an SIC/XE instruction (e.g. MULR) and the opcode integer is that instruction's opcode. The format represents the instruction size: 1, 2, or 3 byte (we will exclude 4–byte instructions in this assignment). The maximum length of a mnemonic string is 5 characters, and therefore can be stored in a 6–byte buffer. The opcode can be stored in 1 byte. You can choose to use an integer or an unsigned char to store it. Your program should insert 1 node into the binary search tree for each instruction in the file. Then, your program should print the contents of the tree to stdout. This should be a level–order representation, with each line representing one level of the tree.

- `programfile` is a file containing the source code for an SIC/XE program. You can assume each line has 1 instruction of the form: `[label] instruction [arguments...]` or `label`

`directive argument`. In the first case, the line contains a standard instruction. In the second case, the line contains a memory allocation directive. Square brackets indicate that the item is optional. Multiple arguments are indicated by dots. For each symbol encountered in the label field, your program should insert 1 node into its hash table. The node should contain the symbol, and its LC value.

Your program should construct the opcode table by inserting the instructions into the BST in the order they appear in the opcode file. Similarly, it should construct the symbol table by inserting the symbols into the hash table in the order they appear in a label field in the program file. Your program should not do any tree rebalancing, and your hash table should be the size specified on the command line.

Your program should print to stdout, the following:

- An `in-order traversal` of the BST. For each node, output the instruction name. For instance, given the opcode file in the example, your program would output: `addr, float, j, lda, ldb, mul, sta`
- The height of the BST. The height of a BST is the maximum length of a path from the root to the deepest node. The length of a path is the number of links. It is defined recursively as follows: The height of a tree with 0 or 1 node is 0. Otherwise, the height is the height of the taller of its two subtrees, plus 1.
- The contents of the hash table, with one array element per line. Each array element is a list of nodes at that index. Each node should display the symbol and LC value. Each line should be of the form: `index: listNode1` $\rightarrow$ `listNode2` $\rightarrow \cdots \rightarrow$ `listNodeN` $\rightarrow$ `NULL`.

Example files:

Opcode File	**Program File**
addr 144 2	start mul val1
ldb 68 3	ldb #2
sta 12 3	addr A B
mul 32 3	float
float 192 1	loop lda #1
lda 0 3	sta val2
j 60 3	j loop
	val1 word 12
	val2 word 0

Remarks:

- Each instruction in the opcode file will have its correct opcode and format listed.
- Each instruction that is utilized in the program file will be one of the instructions listen in the opcode file.
- Register names will be upper–case. All other characters in both files will be lower-case.
- If a line in the program file has a symbol in the label field, then the first character on that line will be the first character of that symbol. Otherwise, the first character on that line will be a tab character (i.e., '/^'), and the second character will be the first character of the name of the instruction or directive.

- All directives will be either 'word', 'resw', or 'resb'. The 'byte' directive won't be used.
- Each instruction/directive name will be at most 5 characters and therefore can be stored in a 6–byte buffer.
- Each symbol will be at most 10 characters and therefore can be stored in an 11–byte buffer.
- There will be no syntax errors in the program file.

Error Handling: For cases not covered by this specification, you may specify and implement a reasonable behavior. Additionally, your program must detect the following fatal errors. In each case, your program should produce a suitable error message to `stderr` and stop.

- A wrong number of command line arguments is provided.
- An input file cannot be opened.

C. Structural Requirements. Your submission must *at least* contain the following files:

1. A C source file with just the `main` function.
2. A header file containing:
   - Structure definition(s) for a BST to store your machine opcode table.
   - Function prototypes for functions related to the machine opcode table, i.e., inserting a node into the table, retrieving an opcode given the instruction name, retrieving the instruction format given the instruction name, and printing the contents of the tree to stdout.
3. A C source file containing implemented functions that correspond to the prototypes above.
4. A header file containing:
   - Structure definition(s) for a hash table to store your symbol table.
   - Function prototypes for functions related to the hash table, i.e., inserting a node into the hash table, retrieving the LC value for a given symbol, and printing the contents of the hash table to stdout.
5. A C source file containing implemented functions that correspond to the prototypes above.
6. The C file hashFunction.c which is provided for you on Blackboard, and already contains a hash function. You must use this hash function in your program.

C. Submission Instructions. Make sure you are logged in to ITSUnix, and your present working directory contains only the files you wish to submit. Use the `turnin-csi402` command to submit all of the .c and .h files, and any file named 'makefile'. Make sure there are no extra .c or .h files in the current directory. If your makefile is called 'Makefile' (with a capital M), then make sure you use a capital M in the turnin command. Instructions for using the `turnin-csi402` command have been provided in Programming Assignment 0, which you should have already submitted.