Section Handout #8

Problem 1: Compiling

Although the interpreter program that appears in this chapter is considerably easier to implement than a complete compiler, it is possible to get a sense of how a compiler works by defining one for a simplified computer system called a *stack machine*. A stack machine performs operations on an internal stack, which is maintained by the hardware, in much the same fashion as the calculator described in Chapter 8. For the purposes of this problem, you should assume that the stack machine can execute the following operations:

```
LOAD #n    Pushes the constant n on the stack.
LOAD var   Pushes the value of the variable var on
           the stack.
STORE va   Stores the top stack value in var without
r          actually popping it.
DISPLAY    Pops the stack and displays the result.
ADD        These instructions pop the top two values
SUB        from the stack and apply the indicated
MUL        operation, pushing the final result back
DIV        on the stack. The top value is the right
           operand, the next one down is the left.
```

Write a function

```
void Compile(FILE *infile, FILE *outfile);
```

that reads expressions from `infile` and writes to `outfile` a sequence of instructions for the stack-machine that have the same effect as evaluating each of the expressions in the input file and displaying their result. For example, if the file opened as `infile` contains

x = 7
y = 5
2 * x + 3 * y

calling `Compile(infile, outfile)` should write the following code to `outfile`:

```
LOAD #7
STORE x
DISPLAY
LOAD #5
STORE y
DISPLAY
LOAD #2
LOAD x
MUL
LOAD #3
LOAD y
MUL
ADD
DISPLAY
```

Problem 2: Implementing Sets with Symbol Tables

Many of the abstract data types we've learned this quarter are closely related. An expression tree is similar to a binary tree and the **graphADT** uses the **setADT** to store its node information. For this problem, we will look at how the methods of a symbol table are well suited to provide the implementation of sets.

Let's say you wanted to have a set of objects that were key/value pairs, similar to ones stored in symbol tables. For example, you might have a set that contains pairs that are students names and their exam scores. No name would be in the set more than once, and in this set, the names would have values associated with them. This problem concerns the implementation of a simplified version of set.h by leveraging the functionality of symtab.h. The set.h that we will be working with is produced on the next page followed by a copy of symtab.h.

The following struct is the way we will define the setCDT for our implementation:

```
/*
 * Type: setCDT
 * ------------
 * This type defines the concrete structure of a set.
 */

struct setCDT {
    symtabADT  table;
};
```

You may assume that the functions **NewSet, FreeSet, SetIsEmpty, AddElement, DeleteElement,** and **IsElement** are already implemented. Your job is to write the remaining two functions:

```
int NElements(setADT set);

setADT Union(setADT s1, setADT s2);
```

Keep in mind that you are now on the implementation side of set module, but the client side of the symtab module.

```
/* set.h */

typedef struct setCDT *setADT;

setADT NewSet(void);
void FreeSet(setADT set);
int NElements(setADT set);
bool SetIsEmpty(setADT set);
void AddElement(setADT set, string key, void *value);
void DeleteElement(setADT set, string key);
bool IsElement(setADT set, string key);
setADT Union(setADT s1, setADT s2);
```

```
/* symtab.h */

typedef struct symtabCDT *symtabADT;

typedef void (*symtabFnT)(string key, void *value,
                          void *clientData);

symtabADT NewSymbolTable(void);
void FreeSymbolTable(symtabADT table);
void Enter(symtabADT table, string key, void *value);
void *Lookup(symtabADT table, string key);
void DeleteSymbol(symtabADT table, string key);
void MapSymbolTable(symtabFnT fn, symtabADT table,
                    void *clientData);

iteratorADT NewIterator(symtabADT table);
bool StepIterator(iteratorADT iterator, void *key);
void FreeIterator(iteratorADT iterator);
```