Section Solutions #7

Problem 1: Mapping over an integer array

```
typedef void (*intmapfn)(int value);                    // ...from .h

/* Function: MapOverIntArray()
 * ---------------------------
 * Given an integer array and a function of type intmapfn, applies
 * the function to each of the elements in the array.
 */

void MapOverIntArray (int array[], int size, intmapfn fn)
{
   int i;

   for (i = 0; i < size; i++) {
      fn (array[i]);
   }
}
```

Problem 2: Client for mapping over an integer array

```
/* Function: PrintInteger()
 * ------------------------
 * This function implements the intmapfn defined for MapOverIntArray()
to
 * print each element in the array.
 */

void PrintInteger (int toPrint)
{
   printf ("%d\n", toPrint);
}
```

The call would look like:

```
MapOverIntArray (array, effSize, PrintInteger);
```

Problem 3: Mapping over a generic array

```
typedef void (*mapfn)(void *value, void *clientData);    // ...from .h

/* Function: MapOverArray()
 * ------------------------
 * Given an array of void *'s and a function of type mapfn, applies
 * the function to each of the elements in the array.  The clientData
 * is passed along to each function call.
 */

void MapOverArray (void **array, int size, mapfn fn, void *clientData)
{
   int i;

   for (i = 0; i < size; i++) {
      fn (array[i], clientData);
   }
}
```

Problem 4: Mapping functions, Iterators, and Symbol Tables

(a) Using a mapping function:

```
string FindLongestKey(symtabADT table)
{
    string longest;

    longest = "";
    MapSymbolTable(ProcessKey, table, &longest);
    return (longest);
}

static void ProcessKey(string key, void *value, void *clientData)
{
    string *pLongest = clientData;

    if (StringLength(key) > StringLength(*pLongest))
      {
        *pLongest = key;
      }
}
```

Note that this approach requires the implementation of `FindLongestKey` to use the `clientData` pointer to keep track of the longest key encountered so far.

(b) Using an iterator:

```
string FindLongestKey(symtabADT table)
{
    string key, longest;
    iteratorADT iterator;

    longest = "";
    iterator = NewIterator(table);
    while (StepIterator(iterator, &key))
      {
        if (StringLength(key) > StringLength(longest))
          {
            longest = key;
          }
      }
    FreeIterator(iterator);
    return (longest);
}
```

(c) It's a matter of personal taste but most programmers would consider the iterator method more intuitive in this case. To use the mapping function, we had to use the client data in sort of a strange way by always changing this value that we're going to eventually return. With an iterator, there was no need to pass around void *'s! In general, mapping functions work best when you want to do the same thing to every element in the table (like increment every value or print every value). Iterators work best when you need to actually access each element (like for comparison). The difference is subtle but as you work with both iterators and mapping functions, you'll start to develop an intuition for which is appropriate.

Problem 5: Unparsing Expressions

```
void Unparse(expressionADT exp)
{
    switch (ExpType(exp)) {
      case IntegerType:
        printf("%d", ExpIntegerValue(exp));
        break;
      case IdentifierType:
        printf("%s", ExpIdentifier(exp));
        break;
      case CompoundType:
        printf("(");
        Unparse(ExpLHS(exp));
        printf(" %c ", ExpOperator(exp));
        Unparse(ExpRHS(exp));
        printf(")");
        break;
    }
}
```