CS106B
Spring 99-00

Handout #34A
May 22, 2000

Section Solutions #8

Problem 1: Compiling

```
/*
 * Function: CompileExp
 * Usage: CompileExp(exp, outfile);
 * --------------------------------
 * This function writes out a set of assembly language instructions
 * to the specified file that, when executed, will leave the result
 * of the expression on the top of the stack.  Like most recursive
 * functions on expressions, the implementation of CompileExp is
 * structured as a simple case dispatch on the expression type.
 */


void CompileExp(expressionADT exp, FILE *outfile)
{
    switch (ExpType(exp)) {
      case IntegerType:
        fprintf(outfile, "LOAD #%d\n", ExpIntegerValue(exp));
        break;
      case IdentifierType:
        fprintf(outfile, "LOAD %s\n", ExpIdentifier(exp));
        break;
      case CompoundType:
        CompileCompound(exp, outfile);
        break;
    }
}


/*
 * Function: CompileCompound
 * Usage: CompileCompound(exp, outfile);
 * -------------------------------------
 * This function handles the compound expression case.
 */


static void CompileCompound(expressionADT exp, FILE *outfile)
{
    char op;

    op = ExpOperator(exp);
    if (op == '=') {
        CompileExp(ExpRHS(exp), outfile);
        fprintf(outfile, "STORE %s\n", ExpIdentifier(ExpLHS(exp)));
    } else {
        CompileExp(ExpLHS(exp), outfile);
        CompileExp(ExpRHS(exp), outfile);
        fprintf(outfile, "%s\n", OperationName(op));
    }
}
```

```
/*
 * Function: OperationName
 * Usage: str = OperationName(op);
 * ------------------------------
 * This function converts an operator character into the assembly
 * language instruction that would implement it.
 */

static string OperationName(char op)
{
   switch (op) {
      case '+': return ("ADD");
      case '-': return ("SUB");
      case '*': return ("MUL");
      case '/': return ("DIV");
      default: Error("Illegal operator in expression");
   }
}
```

#2. Implementing Sets with Symbol Tables

First, using mapping functions:

```
int NElements(setADT set)
{
   int nElems;

   nElems = 0;
   MapSymbolTable(CounterFn, set->table, &nElems);
   return(nElems);
}

void CounterFn(string key, void *value, void *clientData)
{
   int *counter;

   counter = (int *)clientData;
   (*counter)++;
}


setADT Union(setADT s1, setADT s2)
{
   setADT union;

   union = NewSet();
   MapSymbolTable(AddFn, s1->table, union);
   MapSymbolTable(AddFn, s2->table, union);
   return union;
}

void AddFn(string key, void *value, void *clientData)
{
   AddElement((setADT) clientData, key, value);
}
```

Now a solution using iterators:

```
int NElements(setADT set)
{
   int nElems;
   iteratorADT iterator;
   string key;

   nElems = 0;
   iterator = NewIterator(set->table);
   while(StepIterator(iterator, &key))
   {
      nElems++;
   }
   FreeIterator(iterator);
   return(nElems);
}

setADT Union(setADT s1, setADT s2)
{
   setADT union;
   iteratorADT iterator;
   string key;
   void *value;

   union = NewSet();
   iterator = NewIterator(s1->table);
   while(StepIterator(iter, &key))
   {
      value = Lookup(s1->table, key);
      AddElement(union, key, value);
   }
   FreeIterator(iterator);
   iterator = NewIterator(s2->table);
   while(StepIterator(iter, &key))
   {
      value = Lookup(s2->table, key);
      AddElement(union, key, value);
   }
   FreeIterator(iterator);
   return(union);
}
```

Now finally a solution using the foreach idiom:

```
int Nelements(setADT set)
{
   int nElems = 0;
   string key;

   foreach(key in set->table)
   {
      nElems++;
   }
   return(nElems);
}

setADT Union(setADT s1, setADT s2)
{
   setADT union = NewSet();
   string key;
   void *value;

   foreach(key in s1->table)
   {
      value = Lookup(s1->table, key);
      AddElement(union, key, value);
   }
   foreach(key in s2->table)
   {
      value = Lookup(s2->table, key);
      AddElement(union, key, value);
   }
   return(union);
}
```