CS106B Handout #32
Spring 99-00 May 15, 2000

Section Handout #7

Problem 1: Mapping over an integer array

Write the function `MapOverIntArray()` that takes an integer array and a function of type `intmapfn` (defined below) and maps the elements of the array over the function. In other words, it should call the function once for each of the elements in the array. The `intmapfn` is defined as:

```
   typedef void (*intmapfn)(int value);
```

Problem 2: Client for mapping over an integer array

Write a function `PrintInteger()` that can be passed to `MapOverIntArray()` in order to print each of the elements in the array. Given an array `array` and its effective size `effSize`, show what the call to `MapOverIntArray()` would look like.

Problem 3: Mapping over a generic array

Write a function similar to `MapOverIntArray()` called `MapOverArray()` that will work for an array of `void *`'s. In order to give even more flexibility to your clients, have `MapOverArray()` take a `clientData` parameter that will be passed to the mapped function. For this problem, provide the type definition of the mapping function and then implement `MapOverArray()`.

Problem 4: Mapping functions, Iterators, and Symbol Tables

(a) Using the `MapSymbolTable` function defined in the text, write a function

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

which returns the longest key in the symbol table. For example, suppose that you have just entered the following bindings into a symbol table called `elementTable`:

```
        Enter(elementTable, "Hydrogen", "H");
        Enter(elementTable, "Helium", "He");
        Enter(elementTable, "Beryllium", "Be");
        Enter(elementTable, "Boron", "B");
        Enter(elementTable, "Carbon", "C");
        Enter(elementTable, "Klepon", "Kl");
        Enter(elementTable, "Plummerium", "Pl");
```

With these values in the symbol table, calling `FindLongestKey(elementTable)` should return `"Plummerium"`. If there are several keys that are the same length but longer than any other key, `FindLongestKey` may return any of them. For example, if you added the entry

```
        Enter(elementTable, "Harrylaium", "Ha");
```

to the table, your function could return either `"Plummerium"` or `"Harrylaium"` because they have the same length.

(b) Solve the same problem using an iterator instead of a mapping function.

(c) Which method (a or b) seemed more natural? In general, when would you use a mapping function and when would you use an iterator?

Problem 5: Unparsing Expressions

Suppose you have just finished writing the new **`ReadE`** implementation in **`parser.c`** that takes normal operator precedence into account (p. 621 of the text). You want to know if it works or not, but how are you going to test it? You could call **`EvalExp`** on it and see if it comes up with the right answer, but if it doesn't, you won't have much of a clue where it went wrong. You could use the debugger to look at your expression tree, but if it's at all complicated, you'll soon get lost in all those pointers. It may be a better idea to write a function that displays a complete representation of what's in your parse tree. The easiest way to do that is simply to write a function that takes the parse tree representation and converts it back into a fully parenthesized expression.

For example, suppose that you have written such a function and called it **`Unparse`** (which is indeed the standard name for the function that performs this operation). You could then integrate the **`Unparse`** function into the read-eval-print loop as follows:

```
while (TRUE) {
    printf("=> ");
    line = GetLine();
    if (StringEqual(line, "")) break;
    InitScanner(line);
    exp = ReadExp();
    printf("Expression: ");
    Unparse(exp);
    value = EvalExp(exp);
    printf("\nValue: %d\n", value);
}
```

After making this change, the program will unparse the expression before evaluating it, allowing you to infer what the expression tree looks like. For example, after implementing unparse, you should be able to duplicate the following sample run:

```
Expression interpreter
Enter a blank line to stop

=> x = 17↵
Expression: (x = 17)
Value: 17
=> y = 3 * (x + 1)↵
Expression: (y = (3 * (x + 1)))
Value: 54
=>
```

In section, your first job is to write the code for the **`Unparse`** function, which has the following prototype:

```
void Unparse(expressionADT exp);
```