Eric Roberts
CS 106B

Handout #47A
March 2, 2015

Solutions to Section Handout #8

1. Convert an expression to Reverse Polish Notation

```
/*
 * File: ConvertToRPN.cpp
 * ----------------------
 * This program translates an expression from standard infix notation
 * to Reverse Polish form.
 */

#include <iostream>
#include <string>
#include "exp.h"
#include "parser.h"
#include "simpio.h"
#include "tokenscanner.h"
using namespace std;

/* Function prototypes */

string convertToRPN(Expression *exp);

/* Main program */

int main() {
   TokenScanner scanner;
   scanner.ignoreWhitespace();
   scanner.scanNumbers();
   while (true) {
      string line = getLine("=> ");
      if (line == "quit") break;
      scanner.setInput(line);
      Expression *exp = parseExp(scanner);
      cout << convertToRPN(exp) << endl;
      delete exp;
   }
   return 0;
}

/*
 * Function: convertToRPN
 * Usage: string rpn = convertToRPN(exp);
 * --------------------------------------
 * Create a string that displays the expression in Reverse Polish Notation.
 */

string convertToRPN(Expression *exp) {
   if (exp->getType() == COMPOUND) {
      string op = exp->getOperator();
      Expression *lhs = exp->getLHS();
      Expression *rhs = exp->getRHS();
      return convertToRPN(lhs) + " " + convertToRPN(rhs) + " " + op;
   } else {
      return exp->toString();
   }
}
```

2. Constant folding

```
/*
 * Function: foldConstants
 * Usage: Expression *simplified = foldConstants(exp);
 * ---------------------------------------------------
 * This function simplifies an expression by preevaluating constants.
 * To ensure that the new and old expressions can be deleted
 * independently, this code copies the expression structure so
 * that the nodes in the two expression trees are disjoint.
 */

Expression *foldConstants(Expression *exp) {
   ExpressionType type = exp->getType();
   if (type == CONSTANT) {
      return new ConstantExp(exp->getConstantValue());
   } else if (type == IDENTIFIER) {
      return new IdentifierExp(exp->getIdentifierName());
   } else {
      Expression *lhs = foldConstants(exp->getLHS());
      Expression *rhs = foldConstants(exp->getRHS());
      if (lhs->getType() == CONSTANT && rhs->getType() == CONSTANT) {
         EvaluationContext emptyContext;
         int value = exp->eval(emptyContext);
         delete lhs;
         delete rhs;
         return new ConstantExp(value);
      } else {
         return new CompoundExp(exp->getOperator(), lhs, rhs);
      }
   }
}
```

3. Changing the interpreter into a compiler

```
/*
 * Function: compile
 * Usage: compile(infile, outfile);
 * --------------------------------
 * Reads expressions from infile and writes the equivalent stack
 * machine instructions to outfile.
 */

void compile(istream & infile, ostream & outfile) {
   TokenScanner scanner;
   scanner.ignoreWhitespace();
   scanner.scanNumbers();
   scanner.setInput(infile);
   while (scanner.hasMoreTokens()) {
      Expression *exp = readE(scanner, 0);
      compileExp(exp, outfile);
      outfile << "DISPLAY" << endl;
   }
}

/*
 * Implementation notes: compileExp and operatorToInstructionCode
 * --------------------------------------------------------------
 * The compileExp function recursively compiles an expression into
 * stack-machine code; operatorToInstructionCode converts the operator
 * symbol into the name of the instruction that implements it.
 */

void compileExp(Expression *exp, ostream & outfile) {
   ExpressionType type = exp->getType();
   if (type == CONSTANT) {
      outfile << "LOAD #" << exp->toString() << endl;
   } else if (type == IDENTIFIER) {
      outfile << "LOAD " << exp->toString() << endl;
   } else {
      string op = exp->getOperator();
      if (op == "=") {
         compileExp(exp->getRHS(), outfile);
         outfile << "STORE " << exp->getLHS()->toString() << endl;
      } else {
         compileExp(exp->getLHS(), outfile);
         compileExp(exp->getRHS(), outfile);
         outfile << operatorToInstructionCode(op) << endl;
      }
   }
}

string operatorToInstructionCode(string op) {
   if (op == "+") return "ADD";
   if (op == "-") return "SUB";
   if (op == "*") return "MUL";
   if (op == "/") return "DIV";
   error("Illegal operator " + op);
   return "";
}
```