Lesson 30

Objectives: -

- Write a program for foreign language phrase tester

Foreign Language Phrase Tester

Function

This program aids in the learning of a foreign language, by allowing you to translate foreign phrases into your own language, or vice versa. The list of phrases are contained in one or more data files which you specify at the command line.

Programming Issues

The program uses a linked list data structure to store the phrases from the data files, as it is not known at run time how many phrases there will be. The command line arguments must also be parsed to retrieve the filenames of the datafiles to use.

The program uses several "wrapper" functions:

- `Getline()` which wraps around `fgets()`, allowing us to ignore lines in the data files which are blank, or contain comments (comments in this case being lines where the first non-whitespace characters is a '#')

- `Rndm()` which wraps around `rand()`. The construct `rand() % n` is not terribly random, especially with the lower order bits, so we write another function to improve it a little.

Usage

Run the program specifying the data files to use as input, e.g.

```
[paul@localhost paul]$ ./langtest testdata1.dat
testdata2.dat
```

Note that the first two lines of each data file describe the two languages being used; if your data files differ in this respect, the program will behave as if the last data file you specified on the command line contains the correct information.

langtest.c

```
/*

  LANGTEST.C
  =======
  (c) Paul Griffiths 1999
  Email: paulgriffiths@mcmail.com

  Simple foreign language phrase tester

*/


#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include "list.h"


/*  Forward function declarations  */

int GetInput(int argc, char *argv[]);
int Rndm(int lower, int higher);
int Getline(char *buffer, int maxlen, FILE *fp);
```

```
/*  Global variable to hold the names of the two
languages  */

char langs[2][MAX_LINE];


/*  main()  */

int main(int argc, char *argv[]) {

    int   nPhrases, q, lang1, lang2;
    char *phrase[2], buffer[MAX_LINE] = {0};


    /*  Seed random number generator  */

    srand( (unsigned) time(NULL) );


    /*  Get input from files, and store number of
phrases  */

    nPhrases = GetInput(argc, argv);


    /*  Loop until the user types 'q' or 'Q'  */

    while ( buffer[0] != 'q' && buffer[0] != 'Q'
) {


      /*  Pick a random phrase  */

      q = Rndm(0, nPhrases-1);


      /*  Decide which language to test  */

      lang1 = Rndm(0, 1);
      lang2 = lang1 ? 0 : 1;


      /*  Get the relevant phrase, and ask
question  */

      GetItem(q, &phrase[0], &phrase[1]);
      printf("The %s is %s\n", langs[lang1],
phrase[lang1]);
      printf("Enter the %s: ", langs[lang2]);
      fgets(buffer, MAX_LINE-1, stdin);
      printf("The %s is %s\n\n", langs[lang2],
phrase[lang2]);
    }


    /*  Clean up nicely and exit  */
```

```
    FreeList();
    return EXIT_SUCCESS;
}


/*  Gets phrases from specified data files  */

int GetInput(int argc, char *argv[]) {

    int   i, a = 1, length = 0;
    FILE *fp;
    char  buffer1[MAX_LINE], buffer2[MAX_LINE];


    /*  Loop through input files specified
     as command line arguments           */

    while ( argv[a] ) {
     if ( ! (fp = fopen(argv[a], "r")) ) {
         printf("LANGTEST: Error opening %s for
reading...", argv[a]);
         printf("skipping\n");
         continue;
     }

     /*  Determine the two languages used from
         first two lines in the data file
*/

     for ( i = 0; i < 2; i++ ) {
         if (Getline(buffer1,
sizeof(langs[0])/sizeof(langs[0][0])-1, fp)) {
             Trim(buffer1);
             strcpy(langs[i], buffer1);
         }
         else {
             printf("LANGTEST: Error determining
languages.\n");
             exit(EXIT_FAILURE);
         }
     }


     /*  Get pairs of input lines until we have
         no more pairs left in this file
*/

     while ( Getline(buffer1, MAX_LINE-1, fp) ) {
         if ( Getline(buffer2, MAX_LINE-1, fp) )
{
             Trim(buffer1);
             Trim(buffer2);


             /*  Add both phrases to list  */
```

```
            if ( AppendItem(buffer1, buffer2) )
{
                printf("LANGTEST: Failed to
allocate item.\n");
                exit(EXIT_FAILURE);
            }
            ++length;
          }
          else
            break;
      }

      ++a;
      fclose(fp);
    }
    return length;
}


/*  Wrapper function to get a random number -
    rand() % n; is not very random, especially
    with the lower order bits.
*/

int Rndm(int lower, int higher) {

    int range = higher - lower + 1;

    return lower + (int)((double) rand() /
((double)RAND_MAX + 1) * range);
}


/*  Wrapper around fgets(), so we can handle
comments
    (lines that start with '#') and blank lines.
*/

int Getline(char *buffer, int maxlen, FILE *fp) {

    char *temp;


    /*  Loop until we have an acceptable line  */

    do {

      /*  Exit on error  */

      if ( !fgets(buffer, maxlen, fp) )
          return 0;
      temp = buffer;


      /*  Find first non-whitespace character  */

      while ( *temp && isspace(*temp) )
```

```
        ++temp;

     /*  Continue loop if we have a comment or a
blank line  */

    } while ( *temp == 0 || *temp == '#' );

    return 1;
}
```

---

list.h

```
/*

  LIST.H
  ======
  (c) Paul Griffiths, 1999
  Email: mail@paulgriffiths.net

  Interface to linked list operations

*/


#ifndef PG_LIST_H
#define PG_LIST_H


/*  Global macros  */

#define MAX_LINE              (100)


/*  Function declarations  */

int AppendItem(char *foreign, char *english);
int FreeList();
int GetItem(int index, char **phrase1, char
**phrase2);
int Trim(char *buffer);


#endif  /*  PG_LIST_H  */
```

---

list.c

```
/*

  LIST.C
  ======
  (c) Paul Griffiths, 1999
  Email: mail@paulgriffiths.net

  Implementation of linked list functions

*/


#include <stdlib.h>
#include <ctype.h>
#include "list.h"


/*  Our linked list node structure  */

typedef struct node {
    char *phrase[2];
    struct node *next;
} node;


/*  Pointers to start and end of node  */

node *startnode = NULL, *endnode = NULL;


/*  Adds an node onto the end of the list  */

int AppendItem(char *phrase1, char *phrase2) {

    node *tempnode;


    /*  Allocate memory for a new node  */

    if ( !endnode ) {

      /*  If endnode is NULL, we have an empty
          list, so create the first element now.
*/

      if ( ! (startnode = malloc(sizeof(node))) )
          return 1;
      endnode = startnode;
      startnode->next = NULL;
```

```
    }
    else {

      /*  Otherwise, make a new node and make
          our current last node point to it.   */

      if ( ! (tempnode = malloc(sizeof(node))) )
          return 1;
      endnode->next = tempnode;
      endnode = tempnode;
    }


    /*  Initialise node members  */

    endnode->next = NULL;

    if ( ! (endnode->phrase[0] =
malloc(strlen(phrase1)+1)) )
      return 1;
    strcpy(endnode->phrase[0], phrase1);

    if ( ! (endnode->phrase[1] =
malloc(strlen(phrase2)+1)) )
      return 1;
    strcpy(endnode->phrase[1], phrase2);


    return 0;
}


/*  Retrieves from the list the item at the
specified index  */

int GetItem(int index, char **phrase1, char
**phrase2) {

    int n = 0;
    node *tempnode = startnode;


    /*  Cycle through list to specified index  */

    while ( index > n++ )
      tempnode = tempnode->next;


    /*  Update function arguments to point to the
data  */

    *phrase1 = tempnode->phrase[0];
    *phrase2 = tempnode->phrase[1];

    return 0;
}
```

```
/*  Free's all the malloc'ed memory in our list
*/

int FreeList() {

    node *tempnode = startnode;

    while ( tempnode ) {
      free(tempnode->phrase[0]);
      free(tempnode->phrase[1]);
      startnode = tempnode->next;
      free(tempnode);
      tempnode = startnode;
    }

    return 0;
}


/*  Remove trailing whitespace and '\n's from a
string  */

int Trim(char *buffer) {

    int n = strlen(buffer) - 1;

    while ( !isalnum(buffer[n]) )
      buffer[n--] = '\0';

    return 0;
}
```

---

testdata.dat

```
# testdata.dat
#
# Sample LANGTEST data file
#
# These lines are comments

# The first two uncommented lines signify the
# two languages we will be translating. They
# should be in the same order as the pairs of
# phrases which follow

Japanese
English
```

```
# Now that we have specified our languages,
# we can list the pairs of phrases in the
# order specified above. Blank lines are
# ignored by LANGTEST, so we can use them or
# omit them as we please.

O-hayo gozaimasu
Good morning

Konnichi wa
Hello

Konban wa
Good evening

O-yasumi nasai
Good night

Hajimemashite
Pleased to meet you

Sayonara
Goodbye

O-namae wa nan to osshaimasu ka
What is your name?

Tanaka to moshimasu
My name is Tanaka

Sumimasen
Excuse me
```

A Simple Calculator

Function

This program implements a simple calculator, which evaluates expressions such as:

- 1+2
- 3*4+1
- (4+5)*3
- 10 % 4 * 8

Programming Issues

Evaluating expressions such as these is a two part problem:

1. Parsing the input to determine the *operators* (such as +, -, * and /) and the *operands* (such as 1, 34, 112). Operators *operate on* operands.
2. Evaluating the expression according to the order of the operators and operands

Parsing the Input

In this program, parsing the input is relatively simple. Any sequence of digits is an operand, and anything else is an operator. We only allow single character binary operators in this program.

Note: a *binary* operator acts on two operands, such as 1 + 2, 3 / 5, or 3 * 6. A *unary* operator acts on only one operand, such as sqrt 5, or log 234.

In parsing our expressions, therefore, we will step through the input one character at a time, and:

- If we find a digit, we will consider the following sequence of digits to be one operand. We will continue parsing input from the next non-digit character.
- If we find a non-digit character, we will consider it to be an operator. We will continue parsing input from the next character, since all our operators are one character long.
- We will skip all whitespace.

With an expression such as 1 + 2 - 3 + 4, we could parse from left to right and evaluate as we go along, in the following manner:

Step	Action	Progress	Last operator
1	Next token is '1', store this in progress	1	

2	Next token is '+', store this in last operator	1	+
3	Next token is '2', last operator is '+' so add to progress	3	+
4	Next token is '-', store this in last operator	3	-
5	Next token is '3', last operator is -, so subtract from progress	0	-
6	Next token is '+', store this in last operator	0	+
7	Next token is '4', last operator is '+' so add to progress	4	+
8	No more input, so progress is our final value	4	+

However, this is not possible with an expression such as 1 + 2 * 4, as this should evaluate to 9 (as the multiplication should be carried out before the addition) and not 12 (as would happen in a simple left to right evaluation. In other words, some operators have a higher *precedence* than others. In addition, parentheses can be used to alter the "natural" precedence of operators, for example (1 + 2) * 3 which really would equal 12.

Evaluating complex expressions like this is not simple. However, the usual way of presenting expressions such as this, with its precedence rules and parentheses, is not the only way of representing expressions. We can make our life a lot easier by restating the expression in a more straightforward way before trying to evaluate it.

Postfix and Infix Expressions

An expression of the form (3 + 4) * 4 / 2 is known as a *infix expression*, and is the type used in everyday mathematics. It has a series of rules, such as multiply and divide operations are carried out before addition and subtraction operations, and subexpressions

contained within parentheses are evaluated before any others.

A *postfix expression* is much simpler to evaluate. The postfix form of the above expression would be 3 4 + 4 * 2 /. While this may look strange, it's just another way of writing the same expression. In postfix, an operator appears *after* its operands (hence the word **post**fix), rather than in between them, as in an **in**fix expression. There are no parentheses in a postfix expression, and no precedence rules (there is another form of expression called *prefix*, in which an operator comes *before* its operands).

The first thing we do in our calculator therefore, is to convert the input infix expression into a postfix expression. We do this in the following steps:

1. Create a *stack* data structure to temporarily store operators;
2. Read through the infix expression one token at a time from left to right;
3. Output an operand as soon as we come to one (operands always appear in the same order, whether we are using an infix, a postfix, or a prefix expression);
4. When we come to an operator, if it has a higher *precedence* than the operator currently on the top of the stack, then add it to the stack;
5. If the operator has a lower precedence than the operator on top of the stack, then remove operators from the stack one at a time until we run out, or until we come to an operator with a higher precedence.
6. When we run out of input, output any remaining operators still in the stack.

The one exception is that when we come to a closing parenthesis, we output all operators on the stack until we reach its opening parenthesis. In addition, even though we usually only place operators on the stack if the operator currently on the top has a lower precedence, we make a special case when an opening parenthesis is on the top of the stack. The opening parenthesis must have the highest precedence when

parsing an expression, but the lowest precedence when at the top of the stack.

This will all be much easier to understand with a table. In the following example, we will assume '+' and '-' have a precedence of 1, '*' and '/' have a predecence of 2, ')' has a precedence of 3, and '(' has a precedence of 4, and we will convert our previous expression (3 + 4) * 4 / 2:

Step	Action	Output	Stack
1	Read '(', stack empty so place operator on stack		(
2	Read '3', and output	3	(
3	Read '+', lower precedence than '(' but make special case and place on stack	3	(+
4	Read '4', and output	3 4	(+
5	Read ')', higher precedence than '+', so place operator on stack	3 4	(+)
6	Read '*', lower precedence than ')', so remove ')' from stack (do not output parentheses)	3 4	(+
7	'+' is top of stack, lower precedence than '*' but we haven't found the opening parenthesis yet, so remove it and output	3 4 +	(
8	'(' is top of stack, this is the opening parenthesis, so remove it from stack (do not output parentheses)	3 4 +	
9	Stack empty, so place our '*' on stack	3 4 +	*
10	Read '4', and output	3 4 + 4	*
11	Read '/', same precedence as '*' on top of stack, so remove '*' and output	3 4 + 4 *	
12	Stack empty, so place our '/' on stack	3 4 + 4	/

		*	
13	Read '2', and output	3 4 + 4 * 2	/
14	End of input, so output remaining operators on stack	3 4 + 4 * 2 /	

As you can see, our output is the postfix form of our original expression.

Evaluation

Once we have our postfix expression, evaluating it is fairly simple:

1. Create a stack, this time to hold the operands;
2. Read through the input from left to right;
3. If we find an operand, add it to the stack;
4. If we find an operator, remove the last two operands from the stack, perform the operation, and add the value back to the stack;
5. At the end of the input, the one remaining value on the stack is the result of our expression.

Again, in table format:

Step	Action	Stack
1	Read '3', place on stack	3
2	Read '4', place on stack	3 4
3	Read '+', remove last two operands from stack, add them, and place sum on stack	7
4	Read '4', place on stack	7 4
5	Read '*', remove last two operands from stack, multiply them, and place product on stack	28
6	Read '2', place on stack	28 2
7	Read '/', remove last two operands from stack, divide them, and place result on stack	14
8	End of input, so single value remaining on	14

	stack is our expression's value	

Summary

As you can see, evaluating a postfix expression is a lot simpler than evaluating an infix expression. Converting an infix expression to a postfix expression first makes the whole process much easier.

main.c

```
/*
  MAIN.C
  ======
  (c) Copyright Paul Griffiths 2002
  Email: mail@paulgriffiths.net

  Main function for calculator program
*/


#include <stdio.h>
#include <stdlib.h>

#include "eval.h"


#define BUFFERSIZE (256)


int main(void) {
    char input[BUFFERSIZE];

    printf("Enter your expression: ");
    fflush(stdout);
    fgets(input, BUFFERSIZE, stdin);

    printf("Result is: %.2f\n", evaluate(input));

    return EXIT_SUCCESS;
}
```

---

eval.h

```
/*
  EVAL.H
  ==========
  (c) Copyright Paul Griffiths 2002
```

```
  Email: mail@paulgriffiths.net

  Interface to expression evaluation operation
*/


#ifndef PG_EVAL_H
#define PG_EVAL_H


double evaluate(char * infix);


#endif  /*  PG_EVAL_H  */
```

---

eval.c

```
/*
  EVAL.C
  ======
  (c) Copyright Paul Griffiths 2002
  Email: mail@paulgriffiths.net

  Implementation of expression evaluation
operation
*/


#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#include "token.h"


/*  Converts a string based infix numeric
expression
    to a string based postfix numeric expression
*/

char * toPostfix(char * infix, char * postfix) {
    char         buffer[BUFFERSIZE];
    int          stack[STACKSIZE];
    int          top = 0;
    struct token t;

    *postfix = 0;                          /*
Empty output buffer     */

    while ( (infix = GetNextToken(infix, &t)) ) {
        if ( t.type == TOK_OPERAND ) {
```

```
            /*  Always output operands
immediately  */

            sprintf(buffer, "%d ", t.value);
            strcat(postfix, buffer);
        }
        else {

            /*  Add the operator to the stack if:
                 - the stack is empty; OR
                 - if the operator has a higher
precedence than
                   the operator currently on top
of the stack; OR
                 - An opening parenthesis is on
top of the stack   */

            if ( top == 0 ||
oplist[t.value].precedence >

oplist[stack[top]].precedence ||

oplist[stack[top]].value == OP_LPAREN ) {
                stack[++top] = t.value;
            }
            else {
                int balparen = 0;

                /*  Otherwise, remove all
operators from the stack
                    which have a higher
precedence than the current
                    operator. If we encounter a
closing parenthesis,
                    keep removing operators
regardless of precedence
                    until we find its opening
parenthesis.            */

                while ( top > 0 &&
((oplist[stack[top]].precedence >=

oplist[t.value].precedence)
                                        || balparen)
                                    && !(!balparen
                                         &&
oplist[stack[top]].value ==

OP_LPAREN) ) {
                    if ( stack[top] == OP_RPAREN
)
                        ++balparen;
                    else if ( stack[top] ==
OP_LPAREN )
                        --balparen;
                    else {
```

```
                        sprintf(buffer, "%c ",
oplist[stack[top]].symbol);
                        strcat(postfix, buffer);
                    }
                    --top;
                }
                stack[++top] = t.value;
            }
        }
    }


    /*  Output any operators still on the stack
*/

    while ( top > 0 ) {
        if ( oplist[stack[top]].value !=
OP_LPAREN &&
             oplist[stack[top]].value !=
OP_RPAREN ) {
            sprintf(buffer, "%c ",
oplist[stack[top]].symbol);
            strcat(postfix, buffer);
        }
        --top;
    }

    return postfix;
}


/*  Parses a postfix expression and returns its
value  */

double parsePostfix(char * postfix) {
    struct token t;
    double       stack[STACKSIZE];
    int          top = 0;

    while ( (postfix = GetNextToken(postfix, &t))
) {
        if ( t.type == TOK_OPERAND ) {
            stack[++top] = t.value;       /*
Store operand on stack  */
        }
        else {
            double a, b, value;

            if ( top < 2 ) {              /*  Two
operands on stack?  */
                puts("Stack empty!");
                exit(EXIT_FAILURE);
            }

            b = stack[top--];             /*  Get
last two operands   */
            a = stack[top--];
```

```
            switch ( t.value ) {
            case OP_PLUS:                   /*
Perform operation       */
                value = a + b;
                break;

            case OP_MINUS:
                value = a - b;
                break;

            case OP_MULTIPLY:
                value = a * b;
                break;

            case OP_DIVIDE:
                value = a / b;
                break;

            case OP_MOD:
                value = fmod(a, b);
                break;

            case OP_POWER:
                value = pow(a, b);
                break;

            default:
                printf("Bad operator: %c\n",
oplist[t.value].symbol);
                exit(EXIT_FAILURE);
                break;
            }
            stack[++top] = value;        /*  Put
value back on stack  */
        }
    }
    return stack[top];
}


/*  Evaluates an postfix numeric expression  */

double evaluate(char * infix) {
    char postfix[BUFFERSIZE];
    return parsePostfix(toPostfix(infix,
postfix));
}
```

---

token.h

```
/*
  TOKEN.H
  =======
```

```
  (c) Copyright Paul Griffiths 2002
  Email: mail@paulgriffiths.net

  Interface to tokenising operation
*/


#ifndef PG_TOKEN_H
#define PG_TOKEN_H


/*  Macros  */

#define BUFFERSIZE   (256)
#define STACKSIZE    (256)

#define TOK_OPERAND    (0)
#define TOK_OPERATOR   (1)

#define OP_PLUS        (0)
#define OP_MINUS       (1)
#define OP_MULTIPLY    (2)
#define OP_DIVIDE      (3)
#define OP_POWER       (4)
#define OP_MOD         (5)
#define OP_RPAREN      (6)
#define OP_LPAREN      (7)
#define OP_BAD        (99)


/*  Struct definitions  */

struct token {
    int type;
    int value;
};

struct operator {
    char symbol;
    int  value;
    int  precedence;
};


/*  Array of operator descriptions  */

extern struct operator oplist[];


/*  Function declarations  */

char * GetNextToken(char * input, struct token *
t);


#endif  /*  PG_TOKEN_H  */
```

token.c

```
/*
  TOKEN.C
  =======
  (c) Copyright Paul Griffiths 2002
  Email: mail@paulgriffiths.net

  Implementation of tokenising operation
*/


#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

#include "token.h"


/*  Array of operators  */

struct operator oplist[] = { {'+', OP_PLUS,
1},
                             {'-', OP_MINUS,
1},
                             {'*', OP_MULTIPLY,
2},
                             {'/', OP_DIVIDE,
2},
                             {'^', OP_POWER,
3},
                             {'%', OP_MOD,
2},
                             {')', OP_RPAREN,
4},
                             {'(', OP_LPAREN,
5},
                             {0,   0,
0} };


/*  Gets the next token from a string based
numeric
    expression. Returns the address of the first
    character after the token found.
*/

char * GetNextToken(char * input, struct token *
t) {
```

```
    while ( *input && isspace(*input) )  /*  Skip
leading whitespace  */
        ++input;

    if ( *input == 0 )                   /*
Check for end of input   */
        return NULL;

    if ( isdigit(*input) ) {             /*
Token is an operand      */
        t->type  = TOK_OPERAND;
        t->value = strtol(input, &input, 0);
    }
    else {                               /*
Token is an operator     */
        int n = 0, found = 0;

        t->type = TOK_OPERATOR;

        while ( !found && oplist[n].symbol ) {
            if ( oplist[n].symbol == *input ) {
                t->value = oplist[n].value;
                found = 1;
            }
            ++n;
        }

        if ( !found ) {
            printf("Bad operator: %c\n", *input);
            exit(EXIT_FAILURE);
        }
        ++input;
    }
    return input;
}
```

Questions: -

1. Write a program for foreign language phrase tester