```
/*
 * Задание 5.16.  Описать процедуру или функцию, которая печатает дерево-формулу
 * в виде соответствующей формулы.
 */

#include "fp.h"

#define LINE_BUFFER_LENGTH  1024

const operation Operations[] = {
  {'+',  2, " + ", NULL, 0},
  {'-',  2, " - ", NULL, 0},
  {'*',  2, " * ", NULL, 1},
  {'/',  2, " / ", NULL, 1},
  {'\%', 2, " % ", NULL, 1},
  {'^',  2, " ^ ", NULL, 3},
  {'$',  1, "|",    "|",  UNARY_PAREN_OP_NEVER},
  {'~',  1, "-",    NULL, UNARY_PAREN_OP_NESTED | UNARY_PAREN_WHEN_RIGHT},
  {'#',  1, "sgn", NULL, UNARY_PAREN_OP_ALWAYS},
  {'!',  1, NULL,  "!",  UNARY_PAREN_OP_NESTED}
};

const char *strGreeting           ="Enter formulas in reverse polish notation (Ctrl-D to quit):\n"
;
const char *strParsing            ="***Parsing formula:\n%s";
const char *strConstructedTree    ="***Constructed tree:\n";
const char *strUnparsedFormula    ="***Unparsed formula:\n";
const char *strErrOpenFile        ="fp: cannont open file `%s'\n";
const char *strErrMem             ="fp: not enough memory\n";
const char *strErrUnknownOperation="fp: unknown operation `%c'\n";
const char *strErrUnexpectedCharInteractive="fp: column %d--unexpected character `%c'\n";
const char *strErrUnexpectedCharNoninteractive="fp: line %d, col %d--unexpected character `
%c'\n";
const char *strErrParsingExpressionInteractive="fp: unable to parse expression\n";
const char *strErrParsingExpressionNoninteractive="fp: unable to parse expression on line %
d\n";
const char *strErrUnparsingTree   ="fp: unable to unparse formula tree\n";

static char *line;


int main(int argc, char **argv)
{
  int InteractiveMode, lineno;
  char *str;
  tnodeptr ptrRoot;

  /* If user provided filename, try to redirect stdin to that file */
  if ( argc > 1 ) {
    if ( !freopen(*(argv+1), "r", stdin) ) {
      fprintf(stderr, strErrOpenFile, *(argv+1));
      return -1;
    }
    InteractiveMode = 0;
  } else {
    fputs(strGreeting, stdout);
    InteractiveMode = -1;
  }
```

```
  /* Allocate line buffer to read input into */
  if ( !(line = malloc(LINE_BUFFER_LENGTH)) ) {
    fputs(strErrMem, stderr);
    return -1;
  }

  /* Main loop */
  lineno = 1;
  while ( fgets(line, LINE_BUFFER_LENGTH, stdin) ) {
    if ( !InteractiveMode )
      fprintf(stdout, strParsing, line);
    /* Construct tree from the formula */
    ptrRoot = ParseLine(line, lineno++, InteractiveMode);
    if ( ptrRoot ) {
      /* Print out the constructed tree */
      fputs(strConstructedTree, stdout);
      PrintOutTree(ptrRoot);
      /* Unparse tree back into formula and print it out */
      fputs(strUnparsedFormula, stdout);
      if ( str = UnparseTree(ptrRoot) )
        puts(str);
      FreeUnparseBuffer();
    }
    putchar('\n');
  }

  FreeStack();

  return 0;
}



/*
 * Local variables:
 * compile-command: "make -k all && echo && ./fp fp.in"
 * End:
 */
```

```
#include "fp.h"

static void ReportParsingError(int InteractiveMode, int lineno);
static void ReportUnexpectedChar(int InteractiveMode, char c, int lineno, int co
lno);

tnodeptr ParseLine(char *line, int lineno, int InteractiveMode)
{
  char *ptr;
  long op1, op2;
  int  sgn, BailOut, colno;
  tnodeptr ptrNode1, ptrNode2;

  ptr = line;

  BailOut = 0;
  colno = 0;
  while ( *ptr && !BailOut ) {
    colno++;
    if ( isalpha(*ptr) ) {
      ReportUnexpectedChar(InteractiveMode, *ptr, lineno, colno);
      BailOut = -1;

    } else if ( isspace(*ptr) ) {
      /* Skip spaces */
      ptr++;

    } else if ( isdigit(*ptr) || *ptr == '_' ) {
      /* Parse out the number */
      if ( *ptr == '_' ) { /* Minus sign */
        sgn = -1;
        ptr++;
      } else
        sgn = 0;
      op1 = strtol(ptr, &ptr, 0);
      if ( sgn )
        op1 = -op1;
      /* Construct the node and push it onto stack */
      if ( !(ptrNode1 = MakeNode(NT_NUMBER, op1)) || !Push(ptrNode1) )
        BailOut = -1;

    } else {
      int OpID;

      /* Parse out the operation */
      switch ( *ptr ) {
      case '+':
        OpID = ID_PLUS;
        break;
      case '-':
        OpID = ID_MINUS;
        break;
      case '*':
        OpID = ID_MULTIPLICATION;
        break;
      case '/':
        OpID = ID_DIVISION;
        break;
      case '\%': /* modulo */
```

```
        OpID = ID_MODULO;
        break;
      case '$': /* absolute value */
        OpID = ID_ABS;
        break;
      case '~': /* negate */
        OpID = ID_NEGATION;
        break;
      case '^': /* flower power */
        OpID = ID_POWER;
        break;
      case '#': /* signum */
        OpID = ID_SIGNUM;
        break;
      case '!': /* factorial */
        OpID = ID_FACTORIAL;
        break;
      default:
        ReportUnexpectedChar(InteractiveMode, *ptr, lineno, colno);
        BailOut = -1;
        break;
      }

      if ( !BailOut ) {
        /* Create node for this operation */
        if ( ptrNode1 = MakeNode(NT_OPERATION, OpID) )
          /* Pop top node which should be a second operand */
          if ( Pop(&ptrNode1->ptrRight) ) {
            /* Pop next node which should be first operand */
            if ( (Operations+OpID)->ops == 2 ) {
              if ( Pop(&ptrNode1->ptrLeft) ) {
              } else {
                ReportParsingError(InteractiveMode, lineno);
                free( (void *) ptrNode1->ptrRight );
                free( (void *) ptrNode1 );
                BailOut = -1;
              }
            }
          } else {
            ReportParsingError(InteractiveMode, lineno);
            free( (void *) ptrNode1 );
            BailOut = -1;
          }
        else {
          BailOut = -1;
        }
      }

      /* Push constructed subtree back onto stack */
      if ( !BailOut ) {
        if ( !Push(ptrNode1) ) {
          FreeTreeMem(ptrNode1);
          BailOut = -1;
        }
      }

      ptr++;
    }
  }
```

```
  if ( BailOut ) {
    FreeStackedTrees();
    return NULL;
  }

  /* Pop the stack--if expression is correct (or this program is not
   * buggy!) the stack will contain exactly one node which is the root
   * we are about to return */
  if ( !Pop(&ptrNode1) ) {
    ReportParsingError(InteractiveMode, lineno);
    return NULL;
  }

  /* Make sure stack contains no other nodes pushed onto it */
  if ( Pop(&ptrNode2) ) {
    ReportParsingError(InteractiveMode, lineno);
    free( (void *) ptrNode2 );
    FreeStackedTrees();
    return NULL;
  }

  return ptrNode1;
}

static void ReportParsingError(int InteractiveMode, int lineno)
{
  if ( InteractiveMode )
    fputs(strErrParsingExpressionInteractive, stderr);
  else
    fprintf(stderr, strErrParsingExpressionNoninteractive, lineno);

  return;
}

static void ReportUnexpectedChar(int InteractiveMode, char c, int lineno, int co
lno)
{
  if ( InteractiveMode )
    fprintf(stderr, strErrUnexpectedCharInteractive, colno, c);
  else
    fprintf(stderr, strErrUnexpectedCharNoninteractive, lineno, colno, c);

  return;
}




/*
 * Local variables:
 * compile-command: "make -k all && echo && ./fp fp.in"
 * End:
 */
```

```
#include "fp.h"

#define STACK_MEM_CHUNK     16

static tnodeptr *Stack = NULL;
static int StackTop = -1;
static int StackLen = 0;

/* Returns length of stack on success, 0 on failure to grow stack */
int Push(tnodeptr ptrNode)
{
  tnodeptr *ptr;

  /* Grow stack if necessary */
  if ( ++StackTop >= StackLen ) {
    StackLen += STACK_MEM_CHUNK;
    if ( !(ptr = (tnodeptr *) realloc(Stack, StackLen * sizeof(*Stack))) ) {
      StackLen -= STACK_MEM_CHUNK;
      fputs(strErrMem, stderr);
      return 0;
    }
    Stack = ptr;
  }
  /* Push the value onto stack */
  *(Stack + StackTop) = ptrNode;

  return StackTop + 1;
}

/* Returns -1 on success, 0 if stack is empty */
int Pop(tnodeptr *ptrNode)
{
  if ( StackTop >= 0 ) {
    *ptrNode = *(Stack + StackTop--);
    return -1;
  }

  return 0;
}

void FreeStackedTrees(void)
{
  tnodeptr *ptrCurr;

  if ( !Stack )
    return;

  for ( ptrCurr = Stack + StackTop; ptrCurr >= Stack; ptrCurr-- )
    FreeTreeMem(*ptrCurr);

  StackTop = -1;

  return;
}

void FreeStack(void)
{
  FreeStackedTrees();
  free(Stack);
```

```
  Stack = NULL;
  StackLen = 0;
  StackTop = -1;
}



/*
 * Local variables:
 * compile-command: "make -k all && echo && ./fp fp.in"
 * End:
 */
```

```
#include "fp.h"

tnodeptr MakeNode(int type, long value)
{
  tnodeptr ptrNode;

  if ( !( ptrNode = (tnodeptr) malloc( sizeof(tnode) ) ) ) {
    fputs(strErrMem, stderr);
    return NULL;
  }

  ptrNode->type = type;
  ptrNode->value = value;
  ptrNode->ptrLeft = ptrNode->ptrRight = NULL;

  return ptrNode;
}

void FreeTreeMem(tnodeptr ptrRoot)
{
  if (!ptrRoot)
    return;

  FreeTreeMem(ptrRoot->ptrLeft);
  FreeTreeMem(ptrRoot->ptrRight);
  free((void *) ptrRoot);
}

static void SpaceOut(int level)
{
  int i;

  for ( i = 0; i < level; i++ )
    putchar(' ');
}

static void PrintNode(tnodeptr ptrRoot, int level, char LR)
{
  SpaceOut(level);
  printf("%c%d: ", LR, level);
  if ( ptrRoot->type == NT_NUMBER )
    printf("%d\n", ptrRoot->value);
  else
    printf("%c\n", (Operations+ptrRoot->value)->code);
}

static void _PrintOutTree(tnodeptr ptrRoot, int level, char LR)
{
  if ( !ptrRoot )
    return;

  PrintNode(ptrRoot, level, LR);
  if (ptrRoot->ptrLeft)
    _PrintOutTree(ptrRoot->ptrLeft, level+1, 'L');
  if (ptrRoot->ptrRight)
    _PrintOutTree(ptrRoot->ptrRight, level+1, 'R');
}

void PrintOutTree(tnodeptr ptrRoot)
```

```
{
  _PrintOutTree(ptrRoot, 0, 'O');
}


/*
 * Local variables:
 * compile-command: "make -k all && echo && ./fp fp.in"
 * End:
 */
```

```
#include "fp.h"

#define STRING_MEM_CHUNK     16

static char *str = NULL;
static int   strLen = 0;

static const char ParenOpen[]  = "([{<";
static const char ParenClose[] = ")]}>";

#define PAREN_AVAILABLE (sizeof(ParenOpen) - 1)

static int   UnparseNode(tnodeptr ptrNode, int *pos, int PrevPriority, int *Pare
nDepth);
static int   StrOut(char *from, int *pos);
static int   CharOut(char c, int *pos);
static void  CharOut_Restart();
       char *GetUnparseBuffer(void);
       void  FreeUnparseBuffer(void);
static int   itoa(long value, int *pos);
static void  reverse(int start, int end);

char *UnparseTree(tnodeptr ptrRoot)
{
  int i = 0, depth = 0;

  /* Initialize variables */
  CharOut_Restart();

  if ( !ptrRoot )
    return NULL;

  /* Start unparsing of the tree */
  if ( !UnparseNode(ptrRoot, &i, -1, &depth) ) {
    FreeUnparseBuffer();
    return NULL;
  }

  /* Finish off the string with '\0' */
  if ( !CharOut('\0', &i) ) {
    FreeUnparseBuffer();
    return NULL;
  }

  return str;
}

static int UnparseNode(tnodeptr ptrNode, int *pos, int PrevPriority, int *ParenD
epth)
{
  if ( !ptrNode )
    return 0;

  if ( ptrNode->type == NT_OPERATION ) {
    /****************************** OPERATION ******************************/
    int OpenParenPos;
    operationptr ptrOp = (operationptr) Operations+ptrNode->value;

    if ( ptrOp->ops == 1 ) {
```

```
      /********** This is unary operation **********/
      tnodeptr ptrNextNode;
      int InsertParens = 0, BranchParenDepth = 0;

      /* Sanity check */
      if ( !ptrNode->ptrLeft && !ptrNode->ptrRight ||
           ptrNode->ptrLeft &&  ptrNode->ptrRight    ) {
        fputs(strErrUnparsingTree, stderr);
        return 0;
      }
      /* Deal with left-or-right ambiguity */
      if ( ptrNode->ptrLeft )
        ptrNextNode = ptrNode->ptrLeft;
      else
        ptrNextNode = ptrNode->ptrRight;
      /* Output operation's prefix */
      if ( !StrOut(ptrOp->prefix, pos) )
        return 0;
      /* See if we need to insert parentheses */
      if ( ptrOp->priority & UNARY_PAREN_OP_ALWAYS ||
           ptrOp->priority & UNARY_PAREN_OP_NESTED &&
           ( ptrNextNode->type == NT_OPERATION ||
             ptrNextNode->type == NT_NUMBER && ptrNextNode->value < 0 ) )
        InsertParens = -1;
      if ( InsertParens )
        OpenParenPos = (*pos)++;
      /* Output subtree */
      if ( !UnparseNode(ptrNextNode, pos, -1, &BranchParenDepth) )
        return 0;
      *ParenDepth = BranchParenDepth;
      /* Output opening and closing parentheses, if needed */
      if ( InsertParens ) {
        (*ParenDepth)++;
        if ( BranchParenDepth >= PAREN_AVAILABLE )
          BranchParenDepth = PAREN_AVAILABLE - 1;
        if ( !CharOut(*(ParenOpen + BranchParenDepth), &OpenParenPos) ||
             !CharOut(*(ParenClose + BranchParenDepth), pos) )
          return 0;
      }
      /* Finally, output operation's postfix */
      if ( !StrOut(ptrOp->postfix, pos) )
        return 0;

    } else {
      /********** This is binary operation **********/
      int ThisPriority, i;
      int LeftBranchParenDepth = 0, RightBranchParenDepth = 0;
      int RightBranchOpenParenPos, ParenRightOp = 0;

      /* Sanity check */
      if ( !ptrNode->ptrLeft || !ptrNode->ptrRight ) {
        fputs(strErrUnparsingTree, stderr);
        return 0;
      }
      ThisPriority = (Operations+ptrNode->value)->priority;
      /* If this operation's priority is lower than that of the
         previous, we need to parenthesise its operands */
      if ( PrevPriority >= 0 && ThisPriority < PrevPriority )
        OpenParenPos = (*pos)++;
```

```
      /* Output left subtree */
      if ( !UnparseNode(ptrNode->ptrLeft,
                        pos,
                        ThisPriority,
                        &LeftBranchParenDepth) )
        return 0;
      /* Output the operation */
      if ( !StrOut( (Operations+ptrNode->value)->prefix, pos ) )
        return 0;
      /* If right operand is a negative number, or it is a unary
         operation of UNARY_PAREN_WHEN_RIGHT type of parenthesising,
         we need to parenthesise it */
      if ( ptrNode->ptrRight->type==NT_NUMBER && ptrNode->ptrRight->value<0 ||
           ptrNode->ptrRight->type == NT_OPERATION &&
             (Operations + ptrNode->ptrRight->value)->ops == 1 &&
             (Operations + ptrNode->ptrRight->value)->priority & UNARY_PAREN_WHE
N_RIGHT ) {
        ParenRightOp = -1;
        RightBranchOpenParenPos = (*pos)++;
      }
      /* Output right subtree */
      if ( !UnparseNode(ptrNode->ptrRight, pos, ThisPriority, &RightBranchParenD
epth) )
        return 0;
      /* Output parentheses around right operand, if needed */
      if ( ParenRightOp ) {
        i = RightBranchParenDepth++;
        if ( i >= PAREN_AVAILABLE )
          i = PAREN_AVAILABLE - 1;
        if ( !CharOut(*(ParenOpen + i), &RightBranchOpenParenPos) ||
             !CharOut(*(ParenClose + i), pos) )
          return 0;
      }
      /* Choose the larger of the parenthesis depths for this node's depth */
      if ( LeftBranchParenDepth > RightBranchParenDepth )
        i = LeftBranchParenDepth;
      else
        i = RightBranchParenDepth;
      *ParenDepth = i;
      /* Output parentheses around entire operation, if needed */
      if ( PrevPriority >= 0 && ThisPriority < PrevPriority ) {
        (*ParenDepth)++;
        if ( i >= PAREN_AVAILABLE )
          i = PAREN_AVAILABLE - 1;
        if ( !CharOut(*(ParenOpen + i), &OpenParenPos) ||
             !CharOut(*(ParenClose + i), pos) )
          return 0;
      }
    }

  } else {
    /***************************** NUMBER *****************************/
    if ( !itoa(ptrNode->value, pos) )
      return 0;
  }

  return -1;
}
```

```
/* Returns -1 on success, 0 on error (memory allocation failed) */
static int StrOut(char *from, int *pos)
{
  if ( !from )
    return -1;

  while ( *from ) {
    if ( *pos >= strLen ) {
      char *ptr;

      strLen += STRING_MEM_CHUNK;
      if ( !(ptr = (char *) realloc((void *) str, strLen)) ) {
        strLen -= STRING_MEM_CHUNK;
        fputs(strErrMem, stderr);

        return 0;
      }

      str = ptr;
    }

    *(str + (*pos)++) = *from++;
  }

  return -1;
}

/* Returns -1 on success, 0 on error (memory allocation failed) */
static int CharOut(char c, int *pos)
{
  if ( *pos >= strLen ) {
    char *ptr;

    strLen += STRING_MEM_CHUNK;
    if ( !(ptr = (char *) realloc((void *) str, strLen)) ) {
      strLen -= STRING_MEM_CHUNK;
      fputs(strErrMem, stderr);

      return 0;
    }

    str = ptr;
  }

  *(str + (*pos)++) = c;

  return -1;
}

static void CharOut_Restart(void)
{
  str = NULL;
  strLen = 0;
}

char *GetUnparseBuffer(void)
{
  return str;
}
```

```
void FreeUnparseBuffer(void)
{
  if ( str )
    free((void *) str);

  CharOut_Restart();
}

static int itoa(long value, int *pos)
{
  int start;

  if ( value < 0 ) {
    if ( !CharOut('-', pos) )
      return 0;
    value = -value;
  }

  start = *pos;
  do {
    if ( !CharOut(value % 10 + '0', pos) )
      return 0;
    value /= 10;
  } while ( value > 0 );

  reverse(start, *pos - 1);

  return -1;
}

static void reverse(int start, int end)
{
  char c;

  if ( !str )
    return;

  while ( start < end ) {
    c = *(str + start);
    *(str + start++) = *(str + end);
    *(str + end--) = c;
  }

  return;
}



/*
 * Local variables:
 * compile-command: "make -k all && echo && ./fp fp.in"
 * End:
 */
```

```
#ifndef _FP_HEADER_
#define _FP_HEADER_

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

/* Node types */
#define NT_NUMBER               0
#define NT_OPERATION            1

/* Types of parenthesising for operands of unary operations (use only one of the
   operand parenthesising flags on a single operation) */
#define UNARY_PAREN_OP_ALWAYS   1 /* always parenthesise operand */
#define UNARY_PAREN_OP_NESTED   2 /* parenthesise operand only when operand
                                     contains nested operations */
#define UNARY_PAREN_OP_NEVER    4 /* never parenthesise operand */
/* Types of parenthesising for unary operations */
#define UNARY_PAREN_WHEN_RIGHT  8 /* parenthesise operation itself if it is a
                                     right-hand operand of a binary operation */

typedef struct _operation {
  char  code;     /* input code */
  int   ops;      /* number of operands (1 or 2) */
  char *prefix;   /* for binary operations this is operation's output symbol;
                     for unary operations this is output prefix */
  char *postfix;  /* for binary operations this should be NULL, for unary
                     operations this is output postfix */
  int   priority; /* for binary operations this is priority; for unary
                     operations this is bit field indicating when to
                     parenthasise this operation and its operand */
} operation, *operationptr;

extern const operation Operations[];

/* Operation ID's (offsets into Operations[]) */
#define ID_PLUS             0
#define ID_MINUS            1
#define ID_MULTIPLICATION   2
#define ID_DIVISION         3
#define ID_MODULO           4
#define ID_POWER            5
#define ID_ABS              6
#define ID_NEGATION         7
#define ID_SIGNUM           8
#define ID_FACTORIAL        9

typedef struct _tnode {
  int  type;                /* Node type (operation or number) */
  long value;               /* Node value (operation ID or number) */
  struct _tnode *ptrLeft;
  struct _tnode *ptrRight;
} tnode, *tnodeptr;


tnodeptr ParseLine(char *line, int lineno, int InteractiveMode);

tnodeptr MakeNode(int type, long value);
void     FreeTreeMem(tnodeptr ptrRoot);
void     PrintOutTree(tnodeptr ptrRoot);
```

```
char    *UnparseTree(tnodeptr ptrRoot);
char    *GetUnparseBuffer(void);
void     FreeUnparseBuffer(void);

int      Push(tnodeptr ptrNode);
int      Pop(tnodeptr *ptrNode);
void     FreeStackedTrees(void);
void     FreeStack(void);

extern const char *strErrOpenFile;
extern const char *strErrMem;
extern const char *strErrUnknownOperation;
extern const char *strErrUnexpectedCharInteractive;
extern const char *strErrUnexpectedCharNoninteractive;
extern const char *strErrParsingExpressionInteractive;
extern const char *strErrParsingExpressionNoninteractive;
extern const char *strErrUnparsingTree;


#endif /* _FP_HEADER_ */
```