UOU Term Project

Infix expression to Postfix expression.

University of Ulsan IT Convergence
20152262 HONGGEUN JI

1. main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "common.c"

extern char* postfix(char expr[]);
extern char* inputExpression(void);
char* inputExpression(void);     // Take expression from user's input.

int main()
{
    char *expr = inputExpression();
    printf("\nThis is the infix expression.\n");
    printf("%s\n\n", expr);

    char *postfix_expr1 = postfix(expr);
    printf("This is the postfix expression.\n");
    printf("%s\n\n", postfix_expr1);

}

char* inputExpression()
{
    char *expr;
    MALLOC(expr, char, sizeof(char) * MAX_EXPR_SIZE);
    printf("Type the infix expression.\n");
    scanf("%[^\n]s",expr);

    return expr;
}
```

As you can see, this is the main function that program will start at first.
I just added “inputExpression()” function which will take the expression from users.

2. stack_.c

```
#include <stdio.h>
#include <stdlib.h>
#include "common.c"
#define MAX_STACKS 50
#define MALLOC(p, t, s)  if( !(p = (t*)malloc(s))  ){\
                            fprintf(stderr, "Insufficient memory!");\
                            exit(EXIT_FAILURE);\
                            }

int isFullStack(stack_ptr sptr);
       // Check whether stack is full or not.
int isEmptyStack(stack_ptr sptr, int reason);
      // Check whether stack is empty or not.

void push(precedence val, stack_ptr *sptr_top);
      // Push the stack element which value is val to the sptr.
      // sptr is considered as a top in this function.

stack_element pop(stack_ptr *sptr_top);
      // Pop the stack element from the sptr.

int isFullStack(stack_ptr sptr)
{
    int count = 0;
    for(; sptr; sptr = sptr -> link) count++;   // Counting.
    if (count == MAX_STACKS){
        fprintf(stderr, "Stack is full!\n");
        exit(EXIT_FAILURE);
    }
    return 0;
}

int isEmptyStack(stack_ptr sptr, int reason)
{
    if(!sptr){
        switch(reason){
            case 1:
                fprintf(stderr, "There is no lparen in stack! So it
reaches empty while searching.\n");
                exit(EXIT_FAILURE);
            default:
                fprintf(stderr, "The stack is empty!\n");
                exit(EXIT_FAILURE);
        }
    }
```

```
    return 0;
}

void push(precedence val, stack_ptr *sptr_top)
{
    stack_ptr temp = NULL;

    if(!isFullStack(*sptr_top)){
        MALLOC(temp, stack_element, sizeof(stack_element));
        temp -> val = val;
        temp -> link = *sptr_top;
        *sptr_top = temp;
    }

}

stack_element pop(stack_ptr *sptr_top)
      // Parameter is consider as the stack's top.
{
    stack_ptr temp = NULL;
    stack_ptr prev_top = *sptr_top;
      // Save the address where to free.

    if(!isEmptyStack(*sptr_top, 0)){
        MALLOC(temp, stack_element, sizeof(stack_element));
                                // Create stack element to copy.
        temp -> val = (*sptr_top) -> val;
        temp -> link = (*sptr_top) -> link;
        *sptr_top = (*sptr_top) -> link;
        free(prev_top);
    }

    return *temp;
}
```

This is the stack that I composed. Basically this stack ADT is implemented by linked stack

There are basic stack related operations such as pushing, poping, checking the stack status. The description of this stack ADT can be checked in the page at very last.

There are two stack status checking functions. One is “isFullStack” and other is “isEmptyStack”. Those are literally checking whether stack is full or empty. If you see the “isEmptyStack” function, there are switch cases which will tell the developer why the stack

is empty. Only two cases are described but if developer think there are more reasons why the stack is empty, just simply put other cases that might think.

The function “isFullStack” will check while the parameter stpr is NULL. The parameter sptr will follow the linked stack and increasing counting variable value. If counting variable is same as the constant MAX_STACKS, the program will end all the things. The functions “push” and “pop” are just add the stack node to the given sptr linked stack.

Pay attention to the parameter *sptr_top. This is the double pointer and also, this is considered as a top of the linked stack. It means the pointer sptr is considered as a top pointer at least on these functions. As the real linked stack pointer have to be moved since it is considered as top pointer, those functions use a double pointer.

3. postfix.c

```
#include <stdio.h>
#include <stdlib.h>
#include "common.c"
#define MAX_EXPR_SIZE 100
#define MALLOC(p, t, s)  if( !(p = (t*)malloc(s))  ){\
                            fprintf(stderr, "Insufficient memory!");\
                            exit(EXIT_FAILURE);\
                            }

int isp[] = { 0, 19, 12, 12, 13, 13, 13, 0};
            // In-stack precedence.
int icp[] = { 20, 19, 12, 12, 13, 13, 13, 0};
            // Incoming precedence.

//extern stack_ptr initStack(void);
extern void push(precedence val, stack_ptr *sptr_top);
extern stack_element pop(stack_ptr *sptr_top);
extern int isFullStack(stack_ptr sptr);
extern int isEmptyStack(stack_ptr sptr, int reason);

char* postfix(char expr[]);
      // Change infix expression to postfix expression.
precedence getToken(char expr[], char *symbol, int *n);
      // Change symbol to token and returns.
char getOperator(precedence token);
      // Change token to symbol and returns.

char* postfix(char expr[])
{
    precedence token;
    char symbol;        // The character from the expression.
    int n = 0;          // Where to get from the expression.

    char *postfix_expr;
    MALLOC(postfix_expr, char, sizeof(char)* MAX_EXPR_SIZE);
            // Create array to save.
    int m = 0;

    stack_ptr sptr1 = NULL;

    for(token = getToken(expr, &symbol, &n); token != eos; token =
getToken(expr, &symbol, &n)){

        while(token == operand){
            postfix_expr[m++] = symbol;
```

```
            token = getToken(expr, &symbol, &n);
        }

        if(m != 0 && postfix_expr[m-1] != ' ')
                  // To delimite.
            postfix_expr[m++] = ' ';

        if (token == rparen){
            while(sptr1 -> val != lparen){
                if(!sptr1)   // It reaches the end of it's stack.
                    isEmptyStack(sptr1 -> link, 1);
                postfix_expr[m++] = getOperator(pop(&sptr1).val);

                if(m != 0 && postfix_expr[m-1] != ' ')
                        // To delimite.
                    postfix_expr[m++] = ' ';
            }
            pop(&sptr1);        // Pop the lparen.

        }
        else if (token == space);  // Do nothing.
        else {
            while(sptr1 && (isp[sptr1 -> val] >= icp[token])){
                postfix_expr[m++] = getOperator(pop(&sptr1).val);
                if(m != 0 && postfix_expr[m-1] != ' ')
                    postfix_expr[m++] = ' ';
            }
            push(token, &sptr1);

        }

    }
    while(sptr1){
        postfix_expr[m++] = getOperator(pop(&sptr1).val);
        if(m != 0 && postfix_expr[m-1] != ' ')
            postfix_expr[m++] = ' ';
    }

    return postfix_expr;

}

precedence getToken(char expr[], char *symbol, int *n)
{
    *symbol = expr[(*n)++];
    switch(*symbol){
        case '(' : return lparen;
        case ')' : return rparen;
        case '+' : return plus;
```

```
        case '-' : return minus;
        case '/' : return divide;
        case '*' : return times_;
        case '%' : return mod;
        case '\0' : return eos;
        case ' ' : return space;
        default : return operand;
    }
}

char getOperator(precedence token)
{
    switch(token){
        case lparen : return '(';
        case rparen : return ')';
        case plus : return '+';
        case minus : return '-';
        case divide : return '/';
        case times_ : return '*';
        case mod : return '%';
        case space : return ' ';
        default : return '\0';
    }
}
```

The function “postfix()” changes the infix expression to postfix expression. There are three cases that the function operate.

If the token is operand, just save it to the array “postfix_expr”. And if the token is right parenthesis, pop from the linked stack until it reaches the left parenthesis. If the token is operator, compare it with the stack’s operator and push or pop it depending on the situations.

Pay attention to the way how this function delimit the expression element. There are no description of delimiting the element but this function will do by using three cases that I described.

You can see that every cases have entering one space when the process is over. This is just for the expression list. NOT for the “postfix()” function.

4. common.c

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_EXPR_SIZE 100
#define MALLOC(p, t, s)  if( !(p = (t*)malloc(s))  ){\
                            fprintf(stderr, "Insufficient memory!");\
                            exit(EXIT_FAILURE);\
                            }


typedef enum precedence {lparen, rparen, plus, minus,
    times_, divide, mod, eos, operand , space}precedence;

typedef struct stack_element{
    precedence val;
    struct stack_element *link;
}stack_element;

typedef struct stack_element *stack_ptr;
```

This C file contains the structure of stack and precedence.

5. The Result

```
Type the infix expression.
((temp1+temp2-temp3)*23)/(num1-num2)

This is the infix expression.
((temp1+temp2-temp3)*23)/(num1-num2)

This is the postfix expression.
temp1 temp2 + temp3 - 23 * num1 num2 - /

Program ended with exit code: 0
```

6. Description of Linked Stack ADT

ADT Stack

objects : The finite linked lists which have more than 0 node.

functions :

Every stack_element ∈ Stack, stack_ptr is pointer of stack_element,
precedence ∈ enum of precedence

Boolean isFullStack(stack_ptr) ::==
    if(The number of stack_element > Max stack size)
        EXIT
    else
        return FALSE

Boolean isEmptyStack(stack_ptr, int) ::==
    if (stack_ptr == NULL)
        EXIT by case(int)
    else
        return FALSE

void push(precedence, stack_ptr) ::==
    create stack pointer

    if (!isFullStack)
        Allocate memory to stack pointer and
        enter the value and link.
        Then push the stack_element to the top.

stack_element pop(stack_ptr) ::==
    create two stack pointer (one is for stack other is for previous top)

    if (!isEmptyStack)
        Allocate memory to stack pointer and
        copy the stack top's value, link.
        Then free the memory.

    return stack_element