**Report**

**For Data Structures # 7**

**학과 :**

**학번 :**

**이름 :**

컴퓨터공학과

20184071

김도현

**문제**

* 문제
* 다음과 같이 수식을 평가하는 프로그램을 작성하고 테스트하라.
  + 사용자로부터 임의 수식을 입력 받는다.
  + 수식을 후위 식으로 변환한다.
  + 후위 식으로부터 수식 트리를 구성한다.
  + 수식 트리를 평가하고, 그 결과를 출력한다.
  + 가급적 일반적인 계산기와 유사하게 하라. (수치의 자리 수/연산자 등)

1. 문제의 요구사항

- 사용자에게 숫자 및 기호 등을 입력 받는다. (중위 형태)

- 사용자에게 입력 받은 수식인지 정상인지 검사한다

- 정상이 아니면, 계산 실패를 출력한다.

- 정상이면 입력 받은 중위 형태의 식을 후위 형태로 바꾼다

- 바꾼 후위 형태의 식을 가지고 트리를 만든다

- 위에서 생성된 트리로 수식 평가를 진행한다

- 또한, 문제에서 요구한 대로 스택을 사용하여 반복 버전의 중위 후위 탐색 결과를 출력한다.

2. 문제 분석

|  |  |
| --- | --- |
| Input | 중위 형태의 수식 |
| Output | 후위 형태의 수식 및, 계산 결과, 중위 순회 후위 순회 결과 |
| Process | **중위 식 검사 방법**  **1.** 첫번째 문자가 부호인지 확인합니다. 부호 면 올바르지 않은 수식 입니다  **2.** 마지막 문자가 부호인지 확인합니다. 부호 면 올바르지 않은 수식 입니다.  **3.** 1 2 가 정상적으로 수행되었으면, 괄호쌍을 검사합니다. 올바르지 않은 괄호쌍이면, 올바르지 않은 수식입니다.  **4.** 1 2 3 이 문제 없이 수행되었으면, 후위 식으로 변환합니다.  **후위 식 변환 방법**  **1.** 숫자가 나오면 문자열에 숫자 + 공백을 추가합니다 > 2자리 이상 구분을 위함  **2.** 닫는 괄호나 여는 괄호가 아닌 기호가 들어오면 문자에 공백을 추가합니다 > 부호 구분을 위함  **3.** 스택이 비지 않고, 연산자 우선순위를 비교해서 스택에서 pop 을 수행 한 값을 문자열에 넣을지 말지 정합니다.  **3 – 1.** 문자열에 넣었으면, 마지막에 공백을 추가합니다.  **4.** 그 후, 스택에 받아온 문자를 넣습니다.  **5.** 1~4까지 작업이 다 끝났으면, 스택이 비지 않았을 때까지 문자열에 공백 추가 후 부호 추가하는 것을 반복합니다.  **후위 식으로 노드 생성 방법**  생성된 후위 식은 “공백”을 기준으로 나누어 져 있습니다.  **1.** 그렇기 때문에 String 헤더를 포함하고, strtok 함수를 사용하여 공백을 기준으로 받아온 후위 식을 자릅니다.  **2.** 자른 값이 null 이 아닐 때 까지 반복하여 진행합니다. While 문을 통하여 진행하는데, 시작될 때 마다 빈 노드를 생성합니다.  **3.** 자른 문자열이 부호가 아니면, atoi 함수를 사용하여 문자열에서 숫자를 추출합니다. 그 후 생성된 노드의 data 를 atoi 함수의 반환 값으로 설정합니다.  **4.** 자른 문자열이 부호 면, 스택에 있는 값을 두개를 pop 을 수행합니다. 처음 pop 을 수행한 값을 노드의 오른쪽 노드로 설정해주고, 두번째 pop 수행 한 값을 노드의 왼쪽 노드로 설정해줍니다.  **5.** 3번과 4번에서 수행된 노드는 포인터를 가지고 있으므로 스택에 포인터를 넣습니다.  **6.** 자른 문자열이 NULL 이여서 while 문이 종료 되면, 스택엔 루트 노드만 있을 것이므로, 그 노드를 pop 해서 반환합니다.  1 2 \* 33 + 2 – 를 그림으로 그리면 아래와 같습니다.    그 후 이 생성된 트리를 가지고 평가를 진행하면 됩니다.  **트리 평가 진행 방법**  재귀를 사용하여 진행합니다.  **1.** 입력된 root 가 NULL 이면 0을 반환합니다.  **2.** 입력된 root 가 잎 노드면, 숫자이므로, 가지고 있는 숫자를 반환합니다.  **3.** 잎 노드가 아니면, 잎 노드를 찾기 위해 root 의 left 를 재귀적으로 자기 자신 함수를 반환 한 값을 op1으로 할당합니다.  **4.** op1으로 할당 되었으면, root 의 right 를 재귀적으로 자기 자신 함수를 호출하여 반환한 값을 op2 로 할당합니다.  **5.** op1 과 op2 가 할당되었으면, + - \* /등의 부호를 확인하고 그 계산된 값을 반환합니다.  **반복으로 중위 탐색 진행 방법**  **1.** 스택을 생성합니다.  **2.** 무한 반복 문을 실행합니다.  **3.** 무한 반복 문 안에 노드가 null 이 아닐 때 까지 while 문을 수행합니다  **4.** 3번의 while 문을 수행하면서, 스택에 노드를 넣습니다.  **5.** 노드를 넣었으면 노드는 노드가 가진 left 노드로 이동하게 됩니다.  **6.** 스택이 비지 않았으면, 노드는 스택에 있는 값을 pop 울 수행하여 가져옵니다.  **7.** 이 노드가 잎 노드 면, 숫자를 잎 노드가 아니면 기호를 출력합니다.  **8.** 이 작업이 끝났으면, 노드는 노드가 가진 right 로 이동합니다.  **9.** 만약 스택이 비었으면, 무한 반복 문을 종료합니다.  **반복으로 후위 탐색 진행 방법**  **1.** 스택을 생성합니다.  **2.** 방문 된 노드를 저장할 visited 노드를 생성합니다.  **3.** 무한 반복 문을 실행합니다  **4.** 만약 노드가 비지 않았고 방문 된 노드가 아닌 경우 노드를 스택에 넣습니다.  **5.** 스택에 넣었으면, 노드가 NULL 이 아닐 때까지 우측 노드와 좌측 노드를 삽입합니다.  **6.** 삽입이 끝났으면 노드는 좌측 노드로 이동합니다.  **7.** 왜 그렇게 진행하냐 면, 후위 탐색은 좌측 우측 루트 이렇게 출력되는데, 스택은 후입선출 방식을 따르므로, 루트 우측 좌측 순으로 삽입하면, 좌측 우측 루트 순으로 출력되기 때문입니다.  **8.** 4의 if 문이 끝났으면, 스택이 비었 나 확인합니다  **9.** 스택이 비지 않았으면, 노드는 스택에 저장된 값을 반환합니다.  **10.** 만약 노드의 왼쪽이 NULL 이 아니고 노드의 오른쪽이 NULL 이 아니고 방문 된 노드가 아니라면, 우측이 없으므로 좌측 루트 만 출력돼야 하므로, 현재 노드를 스택에 넣고 현재 노드는 스택의 좌측으로 이동합니다.  **11.** 만약 노드의 우측이 NULL 이거나 오른쪽 노드가 방문 된 경우, 데이터를 출력합니다.  잎 노드 면, 숫자 잎 노드가 아니면 부호를 출력합니다. 또한 이 노드는 방문 된 노드이므로 방문 노드는 현재 노드로 설정합니다.  **12.** 10번에서 스택이 비었으면 ( else 문) while 문을 종료합니다. |
| Example | 1. 수식이 올바른 경우  2. 수식이 올바르지 않은 경우  3. 괄호쌍이 올바르고 수식이 올바른 경우  4. 괄호쌍이 올바르고 수식이 올바르지 않은 경우  5. 괄호쌍이 올바르지 않고 수식이 올바른 경우  6. 괄호쌍이 올바르지 않고 수식이 올바르지 않은 경우  5 번 항목의 실행결과 및 검증에서 확인 가능합니다. |

3. 의사 코드를 사용하여 알고리즘 작성

--------------- 중위식 검사 알고리즘 -----------------

procedure is\_exp(Array String) //수식이 올바른지 확인하는 함수

int size <- strlen(String); //strlen 함수를 사용해서 문자열의 크기를 구한다

if (size <= 0) then //문자열의 크기가 0이면

return 0; //0반환

end if

char lastsym <- String[size - 1]; //마지막 문자열은 스트링의 길이 - 1 번째 인덱스

if (is\_sign\_char(lastsym) or is\_sign\_char(String[0]) ) then //처음 또는 마지막 문자가 부호면

// + 6 \* 꼴이므로 변환불가

return 0; //0반환

end if

//문자열의 앞과 뒤 확인했으면 괄호쌍 확인

Stack stack; //확인을 위해선 스택이 필요하기 때문에 스택을 생성한다.

initStack(&stack); //생성한 스택을 초기화한다.

char temp; //임시로 사용할 char 변수, if 문에 사용된다.

for (int i <- 0 to i < size by i++) do //문자열의 길이만큼 반복한다

temp <- String[i];

if (is\_empty(stack) and temp = ')') then //스택이 빈 상황인데 ) 가 들어가면 스택은 ) 꼴이 된다.

return 0; //이는 곧 맞지 않은 괄호쌍을 의미하기에 false 로 만든다.

endif

if (temp = ')') then //위의 if 문이 수행이 안되므로 해당 라인은 수행이된다.

//들어온 값이 ) 경우

pop(&stack); //스택엔 (....(( 꼴로 저장되어있기에 마지막 ( 를 지운다

else if(temp = '(') then

push(&stack, temp); //( 이 들어오는 경우는 항상 스택에 저장한다.

End if

repeat

if (!is\_empty(stack)) then

return 0; //스택에 값이 있으면 올바르지않은 수식

end if

return 1; //위에서 모든 식에서 return 이 수행 안됬으면 올바른 수식이므로 1 반환

end is\_exp

--------------------중위식 검사 알고리즘 끝-------------------

----------------------후위식 변환 알고리즘----------

procedure postfix(char \*String) //후위로 변환된 스트링을 반환하는 함수

Stack stack; //후위식 변환을 위해 스택 사용

initStack(&stack);

char \*temp <- (char \*)malloc(sizeof(char) \* 200); //반환할 스트링, malloc 으로 동적생성

int index <- 0;

char sym;

int len <- strlen(String); //스트링의 길이를 잰 값 반환

for size <- 0 to size < len by size <- size + 1 do //스트링의 길이 만큼 반복

sym <- String[size];

int token <- pie(sym);

if (sym = ')') then

char left;

while ((left <- pop(&stack)) != '(') do

temp[index] <- ' ';

temp[++index] <- left;

temp[++index] <- NULL;

//스트링의 종료는 NULL 로 선언되므로 마지막에 NULL

repeat

else if (pie(sym) = -1) then

temp[index] <- sym;

temp[++index] <- NULL;

//스트링의 종료는 NULL 로 선언되므로 마지막에 NULL

else

if (sym != '(' and sym != ')') then //그 외의 경우 중 ( 와 ) 가 아니면

temp[index] <- ' '; //기호와 숫자 구분해야 하므로 공백 추가

temp[++index] <- NULL;

//스트링의 종료는 NULL 로 선언되므로 마지막에 NULL

endif

while ((is\_empty(stack) != 1) and (pis(peek(stack)) >= pie(sym))) do //for문이 끝나면, 스택이 빌때까지 반복

temp[index] <- pop(&stack); //문자가 들어오는 경우 이므로 pop 연산 후 문자열 뒤에 추가

temp[++index] <- ' '; //마지막에 공백 추가

temp[++index] <- NULL;

//스트링의 종료는 NULL 로 선언되므로 마지막에 NULL

repeat

push(&stack, sym);

endif

repeat

while (is\_empty(stack) != 1) do //스택이 비지 않을떄까지

temp[index] <- ' '; //공백 추가 후

temp[++index] <- pop(&stack); //부호 추가

temp[++index] <- NULL;

//스트링의 종료는 NULL 로 선언되므로 마지막에 NULL

repeat

return temp;

end postfix

------------------후위식 변환 알고리즘 끝---------------

-------------- 루트 노드 반환------------------

Procedure createRoot(charArray String)

/\*

1. 문자열이 끝날때 까지 반복한다

2. 피연산자가 나오면 push 를 수행한다

3. 연산자가 나오면 pop 을 두번 수행한다

4. 두번 수행한 pop 을 가지고 연산자가 부모고 오른쪽 자식 노드는 처음 수행한 pop 에 할당

>> if 문이 처음에 무조건 2번 이상 수행되므로 pop 했을때 포인터가 아닌 경우는 없다!!

5. 두번째 수행한 pop 은 왼쪽 자식 노드에 할당

6. 그 후 생성된 노드를 스택에 다시 삽입

6 - 1. 스택의 데이터 타입은 int 형이므로 생성된 주소값을 int 형으로 형변환후 삽입

7. 반복문이 끝나면 pop 을 수행하여, 노드를 반환한다.

7 - 1. 스택의 데이터 타입은 int 형인데, pop 을 하면 int 형이므로 TreeNode\* 형변환후 반환

\*/

Stack stack; //root 노드를 저장하기 위한 stack 선언

initStack(&stack); //스택 초기화

TreeNode \*node;

char split <- split(String, " "); //입력받아온 String 은 공백을 기준으로 분리해야 하므로 공백 기준으로 분리

while (split != NULL) do //split 포인터가 NULL 이 아닐때 까지 반복

node <- createNode(); //빈 노드를 생성한다

if (!is\_sign\_String(split)) then //split 포인터 즉, 문자열이 부호가 아니면

int data <- split 을 int 로 변환;

setData(node, data);

//node->data <- data; //추출된 숫자를 node 의 data 로 할당

else

setRight(node, pop(&stack)); //문자열이 부호면 스택에 저장되어있는 맨 마지막 값을 가져와서

//right 에 할당,스택 반환값이 달라서 TreeNode 형변환

setLeft(node, pop(&stack));//한번 더 pop 을 수행하여 left 에 할당, 스택 반환값이 달라서 TreeNode 형변환

setData(node, split[0]);//그 후 부호를 data 에 할당

end if

push(&stack, node); //작업이 끝났으면 스택에 넣는다. 데이터 타입이 다르므로 int 로 형변환

repeat

return pop(&stack); //while 문이 끝나면 루트 노드만 남아있으므로 pop 을 하여 루트 노드 반환

end createRoot

--------------루트 노드 반환 알고리즘끝-----------------

------------ 반복버전 중위탐색 -----------------------------

procedure inorder\_iter(TreeNode \*root)

TreeNode \*curr <- root;

Stack s;

Stack\_Ptr stack <- &s;

initStack(stack);

while (1) do

while (curr != NULL) do

push(stack, curr); //왼쪽으로 이동하면서 모든 노드를 스택에 삽입,

curr <- curr의 left;

repeat

if (!is\_empty(\*stack)) then

curr <- pop(stack);

if (is\_leaf(curr)) then //잎노드만 숫자를 가지고 있으므로

print("[%d]", curr->data); //숫자로 출력

else

printf("[%c]", curr->data) //문자로 출력

end if

//데이터는 curr 의 오른쪽으로 이동 한다.

curr <- curr 의 right;

else

break;

endif

repeat

end inorder\_iter

--------------- 반복버전 중위탐색 끝 ---------------------

----------반복 버전 후위 탐색 시작 ----------------------

procedure postorder\_iter(TreeNode \*root) {

TreeNode \*curr <- root;

Stack s;

Stack\_Ptr stack <- &s;

initStack(stack);

TreeNode \*visited <- NULL;

while (1) do

if (curr and curr != visited) then //현재 노드가 NULL 이 아니고 방문되지 않은것이면

push(stack, curr); //curr 을 스택에 넣는다.

//후위 탐색은 좌측 우측 루트 이렇게 출력된다.

//스택은 먼저 들어간게 마지막에 나오므로 후위 탐색을 진행할려면

//루트 우측 좌측 노드 이렇게 삽입을 해야한다.

while (curr != NULL) { //curr 이 NULL 이 아닐때 까지 반복한다.

//좌측 우측 루트로 나와야 하므로

if (curr->right) then

push(stack, curr 의 right); //스택에 넣는다

end if //루트는 삽입되었으므로 우측 삽입

if (curr->left) then //curr 의 left 가 NULL 이 아니면

push(stack, curr 의 left); //스택에 넣는다

end if //루트 우측 삽입되었으므로 좌측 삽입

curr <- curr 의 left; //curr 은 좌측 노드로 이동한다.

repeat

end if

if (!is\_empty(\*stack)) then //스택이 비지 않았으면 수행

curr <- (TreeNode \*)pop(stack); //스택 반환값이 달라서 TreeNode 형변환

//예외 처리, 왼쪽 노드는 존재하는데 오른쪽 노드는 없는경우

//또한 왼쪽 노드가 방문되지 않은 경우

if (curr 의 left and curr 의 right = NULL and curr 의 left != visited) then

//현재 노드를 삽입하고

push(stack, curr); //루트 우측 좌측인데 우측이 없으므로 루트 좌측

//좌측 노드로 이동한다

curr <- curr 의 left;

end if

if (curr->right = NULL or curr->right = visited) then

//오른쪽 노드가 NULL 이거나 오른쪽 노드가 방문된 경우

//후위탐색은 좌측 우측 루트 (본인) 이다.

//위에서까진 좌측 노드가 다 된 경우라 현재는

//우측노드가 완료됬거나 없으면 출력된다

if (is\_leaf(curr)) then //잎노드만 숫자를 가지고 있으므로

printf("[%d]", curr 의 data); //숫자 출력

else

printf("[%c]", curr->data);

end if

//현재 노드를 완료된 노드로 설정

visited <- curr;

end if

else

break;

end if

repeat

end postorder\_iter

---------------반복버전 후위 탐색 끝---------------

4. C 언어로 작성한 코드

#include <stdio.h>

#define MAXSIZE 100

#include <string.h> //strtok 를 사용하기 위한 string 헤더 포함

#include <math.h> //pow 를 사용하기 위한 pow 헤더 포함

typedef int element; //char , int 형, 포인터를 저장할 변수, int 형이 포인터값 (2byte), char 값(2byte) 보다 크므로 저장 가능

typedef struct {

element data[MAXSIZE]; //스택 구조체 정의

int top;

}Stack; //스택으로 데이터 타입 선언

typedef struct tree { //이진트리 구조체 생성

element data; //이진트리는 데이터

struct tree \*left; //왼쪽 자식트리

struct tree \*right; //오른쪽 자식트리를 갖는다.

}TreeNode;

typedef Stack\* Stack\_Ptr; //스택 포인터 정의

//함수 원형 정의부 시작

void initStack(Stack\_Ptr StackPointer);

int is\_empty(Stack stack);

element pop(Stack\_Ptr StackPointer);

void push(Stack\_Ptr StackPointer, element data);

int is\_full(Stack stack);

element peek(Stack stack);

int pis(char sym);

int pie(char sym);

char \* postfix(char \* String);

int is\_sign\_String(char \* sign);

TreeNode \* createNode();

void \* setData(TreeNode \* root, int data);

void \* setLeft(TreeNode \* root, TreeNode \* left);

void \* setRight(TreeNode \* root, TreeNode \* right);

TreeNode \* createRoot(char \* String);

void inorder(TreeNode \* root);

int is\_leaf(TreeNode \* root);

int eval(TreeNode \* root);

void deleteTree(TreeNode \* root);

void inorder\_iter(TreeNode \* root);

void postorder\_iter(TreeNode \* root);

int is\_exp(char \*String);

//함수 원형 정의부 끝

int main() {

/\*

알고리즘은 다음과 같다.

1.문자열을 받아온다

2.문자열에서 숫자, 부호를 판별한다

3.판별된 문자를 공백을 기준으로 분리한다

3 - 1. 예를들여 34 + 4 면 344+ 가 되므로, 34 4 + 로 구분하기 위함

4. 공백을 기준으로 strtok 함수를 사용하여 문자를 분리한후, 부호인지 확인한다

5. 부호가 아니면, atoi 함수를 사용하여, 숫자로 만들고 스택에 넣는다

6. 부호면, 스택에서 pop 을 두번 수행 한후, 트리로 만들고 스택에 넣는다.

7. 이 작업을 반복하고 root 노드를 반환하면 된다.

8. 반환된 root 노드를 이용하여 후위평가를 진행한다

9. 그 후, 메모리 누수 방지를 위해 반환된 노드 등을 free 처리한다.

\*/

int restart;

do {

char String[100];

printf("========================\n");

printf("중위식을 입력하세요 : ");

scanf("%s", String);//수식 입력 (중위식)

if (!is\_exp(String)) { //올바르지 않은 수식이면

printf("잘못된 수식입니다.\n"); //오류를 출력하고

printf("========================\n");

printf("다시 수행하시겠습니까? (1. 예, 기타. 아니오) : ");

scanf("%d", &restart);//다시 계산할지확인을 한다. 1을 입력하면 재시작

continue; //continue 를 사용하여 다음 while 문 수행

}

printf("입력한 중위식 : %s\n", String); //입력한 중위식 출력

char \*posted = postfix(String); //입력받은 중위식을 후위식으로 변환한다.

printf("변환된 후위식 : %s\n", posted);//변환된 후위식 출력

TreeNode \*root = createRoot(posted); //기술한 알고리즘 토대로 root 를 만든 후 반환한다. > 후위 표기법으로부터 수식 트리 생성

printf("반복 버전 중위 순회 결과 : ");

inorder\_iter(root); //트리 중위 순회

printf("반복 버전 후위 순회 결과 : ");

postorder\_iter(root);

int res = eval(root);// 평가된 후위식값

printf("계산 결과 : %d\n", eval(root)); //계산 결과 출력

printf("========================\n");

printf("다시 수행하시겠습니까? (1. 예, 기타. 아니오) : ");

scanf("%d", &restart);//다시 계산할지확인을 한다. 1을 입력하면 재시작

free(posted); //메모리 누수 방지를 위해 posted free

deleteTree(root); //메모리 누수 방지를 위해 root free

} while (restart == 1); //1 을 입력한 경우 재시작

return 0;

}

void initStack(Stack\_Ptr StackPointer) { //스택 초기화, 원본 데이터 수정해야 하므로 StackPtr 을 받아옴

StackPointer->top = -1; //스택의 top 을 -1 로 초기화 해준다

}

int is\_empty(Stack stack) { //스택이비었나 확인, 원본데이터 수정하면 안되므로 Stack 을 받아옴

return (stack.top == -1); //top 이 -1 이면 1, -1이 아니면 0 반환

}

element pop(Stack\_Ptr StackPointer) { //스택 pop 연산, 원본 데이터 수정해야하므로 stackPtr 을 받아옴

if (is\_empty(\*StackPointer)) { //스택이 비었으면 pop 이 수행이 불가하다

printf("더이상 pop 을 수행할수 없습니다."); //그러므로 오류메세지 출력후

exit(1); //프로그램 종료

}

//스택이 안비었으면 정상적으로 pop 이 수행된다

element temp = StackPointer->data[StackPointer->top]; //반환할 값을 할당

StackPointer->top -= 1; //top 을 -1 을 해준다 > 값이 삭제되는것 처럼 보임

return temp; //값 반환

}

void push(Stack\_Ptr StackPointer, element data) { //push 연산, 원본 데이터 수정해야 하므로 stackptr 을 받아옴

if (is\_full(\*StackPointer)) { //스택이 꽉찼으면 push 수행불가

printf("더이상 push 를 수행 할 수 없습니다.");

exit(1);//프로그램 종료

}

StackPointer->top += 1; //스택이 안꽉찼으면 top 을 1 증가시키고

StackPointer->data[StackPointer->top] = data; //증가시킨 인덱스에 data 를 삽입

}

int is\_full(Stack stack) { //스택이 꽉찼나 확인하는 함수

return(stack.top == MAXSIZE - 1); //배열의 최대 사이즈와 top 이 같으면 1, 아니면 0 반환

// > 스택은 top 을 기준으로 데이터 삽입을 판단하기 때문

}

element peek(Stack stack) { //peek 함수, 스택의 상단 값 확인, 원본 수정되면 안되므로 stack 을 받아옴

if (is\_empty(stack)) { //스택이 비어있으면 peek 이 불가하다

printf("peek 수행 불가!");

exit();

}

return (stack.data[stack.top]); //그게 아니면 최상단 값을 반환한다.

}

int pis(char sym) { //스택 안에서 순위 검사

switch (sym) {

case '(': return 0; //여는 괄호가 마지막

case ')': return 3; //닫는 괄호가 두번째

case '+':case'-': return 1; //더하기 빼기가 네번째

case '\*':case'%':case'/': return 2; //곱하기 나누기 나머지가 세번째

case '^': return 4; //제곱이 첫순위

}

return -1;

}

int pie(char sym) { //잘 보면 59 번 60번 라인과 68번 69번 라인의 리턴값이 바뀐것을 볼 수 있다.

switch (sym) { //스택 밖에서 순위 검사

case '(': return 3; //여는 괄호가 두번째

case ')': return 0; //닫는 괄호가 마지막

case '+':case'-': return 1; //더하기 빼기가 네번째

case '\*':case'%':case'/': return 2; //곱하기 나누기 나머지가 세번째

case '^': return 4;//제곱이 제일 첫순위

}

return -1;

}

char\* postfix(char \*String) { //후위로 변환된 스트링을 반환하는 함수

Stack stack; //후위식 변환을 위해 스택 사용

initStack(&stack);

char \*temp = (char \*)malloc(sizeof(char) \* 200); //반환할 스트링, malloc 으로 동적생성

int index = 0;

char sym;

int len = strlen(String); //스트링의 길이를 잰 값 반환

for (int size = 0; size < len; size++) { //스트링의 길이 만큼 반복

sym = String[size];

int token = pie(sym);

if (sym == ')') {

char left;

while ((left = pop(&stack)) != '(') {

temp[index] = ' ';

temp[++index] = left;

temp[++index] = NULL;

//스트링의 종료는 NULL 로 선언되므로 마지막에 NULL

}

}

else if (pie(sym) == -1) {

temp[index] = sym;

temp[++index] = NULL;

//스트링의 종료는 NULL 로 선언되므로 마지막에 NULL

}

else {

if (sym != '(' && sym != ')') { //그 외의 경우 중 ( 와 ) 가 아니면

temp[index] = ' '; //기호와 숫자 구분해야 하므로 공백 추가

temp[++index] = NULL;

//스트링의 종료는 NULL 로 선언되므로 마지막에 NULL

}

while ((is\_empty(stack) != 1) && (pis(peek(stack)) >= pie(sym))) { //for문이 끝나면, 스택이 빌때까지 반복

temp[index] = pop(&stack); //문자가 들어오는 경우 이므로 pop 연산 후 문자열 뒤에 추가

temp[++index] = ' '; //마지막에 공백 추가

temp[++index] = NULL;

//스트링의 종료는 NULL 로 선언되므로 마지막에 NULL

}

push(&stack, sym);

}

}

while (is\_empty(stack) != 1) { //스택이 비지 않을떄까지

temp[index] = ' '; //공백 추가 후

temp[++index] = pop(&stack); //부호 추가

temp[++index] = NULL;

//스트링의 종료는 NULL 로 선언되므로 마지막에 NULL

}

return temp;

}

int is\_sign\_String(char \*sign) { //문자열이 부호인지 확인

return !(strcmp(sign, "+") != 0 && strcmp(sign, "-") != 0 && strcmp(sign, "/") != 0 && strcmp(sign, "\*") != 0 && strcmp(sign, "^") && strcmp(sign, "%") != 0); //부호가 아니면 0

//부호면 1 반환

}

int is\_sign\_char(char sym) { //문자가 부호인지 확인

return (sym == '+' || sym == '-' || sym == '\*' || sym == '/' || sym == '^' || sym == '%'); //부호면 1 부호가 아니면 0 반환

}

int is\_exp(char \*String) { //수식이 올바른지 확인하는 함수

int size = strlen(String); //strlen 함수를 사용해서 문자열의 크기를 구한다

if (size <= 0) { //문자열의 크기가 0이면

return 0; //0반환

}

char lastsym = String[size - 1]; //마지막 문자열은 스트링의 길이 - 1 번째 인덱스

if (is\_sign\_char(lastsym) || is\_sign\_char(String[0]) ){ //처음 또는 마지막 문자가 부호면

// + 6 \* 꼴이므로 변환불가

return 0; //0반환

}

//문자열의 앞과 뒤 확인했으면 괄호쌍 확인

Stack stack; //확인을 위해선 스택이 필요하기 때문에 스택을 생성한다.

initStack(&stack); //생성한 스택을 초기화한다.

char temp; //임시로 사용할 char 변수, if 문에 사용된다.

for (int i = 0; i < size; i++){ //문자열의 길이만큼 반복한다

temp = String[i];

if (is\_empty(stack) && temp == ')') { //스택이 빈 상황인데 ) 가 들어가면 스택은 ) 꼴이 된다.

return 0; //이는 곧 맞지 않은 괄호쌍을 의미하기에 false 로 만든다.

//break; //더이상 볼 가치가 없으므로 루프를 종료한다.

}

if (temp == ')') { //위의 if 문이 수행이 안되므로 해당 라인은 수행이된다.

//들어온 값이 ) 경우

pop(&stack); //스택엔 (....(( 꼴로 저장되어있기에 마지막 ( 를 지운다

//(....( 이런식으로 말이다.

}

else if(temp == '('){

push(&stack, temp); //( 이 들어오는 경우는 항상 스택에 저장한다.

}

}

if (!is\_empty(stack)) {

return 0; //스택에 값이 있으면 올바르지않은 수식

}

return 1; //위에서 모든 식에서 return 이 수행 안됬으면 올바른 수식이므로 1 반환

}

TreeNode \*createNode() { //노드를생성하는 함수

TreeNode \*temp = (TreeNode \*)malloc(sizeof(TreeNode)); //노드 동적 생성

temp->data = 0; //데이터와 왼쪽 오른쪽은 기본값 0 과 NULL 로 설정

temp->left = NULL;

temp->right = NULL;

return temp; //생성된 temp 포인터 반환

}

void \*setData(TreeNode \*root, int data) { //데이터를 설정 하는 함수

root->data = data; //받아온 root 의 data 필드를 받아온 data 로 설정

}

void \*setLeft(TreeNode \*root, TreeNode \*left) { //왼쪽 자식 노드를 설정하는 함수

root->left = left; //받아온 root 의 left 필드를 받아온 left 포인터로 설정

}

void \*setRight(TreeNode \*root, TreeNode \*right) { //오른쪽 자식 노드를 설정하는 함수

root->right = right; //받아온 root 의 right 필드를 받아온 right 포인터로 설정

}

TreeNode\* createRoot(char\* String) {

/\*

1. 문자열이 끝날때 까지 반복한다

2. 피연산자가 나오면 push 를 수행한다

3. 연산자가 나오면 pop 을 두번 수행한다

4. 두번 수행한 pop 을 가지고 연산자가 부모고 오른쪽 자식 노드는 처음 수행한 pop 에 할당

>> if 문이 처음에 무조건 2번 이상 수행되므로 pop 했을때 포인터가 아닌 경우는 없다!!

5. 두번째 수행한 pop 은 왼쪽 자식 노드에 할당

6. 그 후 생성된 노드를 스택에 다시 삽입

6 - 1. 스택의 데이터 타입은 int 형이므로 생성된 주소값을 int 형으로 형변환후 삽입

7. 반복문이 끝나면 pop 을 수행하여, 노드를 반환한다.

7 - 1. 스택의 데이터 타입은 int 형인데, pop 을 하면 int 형이므로 TreeNode\* 형변환후 반환

\*/

Stack stack; //root 노드를 저장하기 위한 stack 선언

initStack(&stack); //스택 초기화

TreeNode \*node;

char \*split = strtok(String, " "); //입력받아온 String 은 공백을 기준으로 분리해야 하므로 공백 기준으로 분리

while (split != NULL) { //split 포인터가 NULL 이 아닐때 까지 반복

node = createNode(); //빈 노드를 생성한다

if (!is\_sign\_String(split)) { //split 포인터 즉, 문자열이 부호가 아니면

int data = atoi(split); //그 값은 숫자이므로 atoi 함수로 숫자 추출

setData(node, data);

//node->data = data; //추출된 숫자를 node 의 data 로 할당

}

else {

setRight(node, (TreeNode \*)pop(&stack)); //문자열이 부호면 스택에 저장되어있는 맨 마지막 값을 가져와서

//right 에 할당,스택 반환값이 달라서 TreeNode 형변환

setLeft(node, (TreeNode \*)pop(&stack));//한번 더 pop 을 수행하여 left 에 할당, 스택 반환값이 달라서 TreeNode 형변환

setData(node, split[0]);//그 후 부호를 data 에 할당

}

push(&stack, (int)node); //작업이 끝났으면 스택에 넣는다. 데이터 타입이 다르므로 int 로 형변환

split = strtok(NULL, " "); //다음 문자열로 이동

}

return (TreeNode \*)pop(&stack); //while 문이 끝나면 루트 노드만 남아있으므로 pop 을 하여 루트 노드 반환

//스택 반환값이 달라서 TreeNode 형변환

}

void inorder(TreeNode \*root) { //중위 탐색, 좌노드 루트노드 우노드 순 방문

if (root != NULL) {

inorder(root->left);

if (!is\_leaf(root)) { //잎노드에만 숫자가 있다

printf("[%c]", root->data); //잎노드가 아니면 문자 출력

}

else {

printf("[%d]", root->data); //잎노드면 숫자 출력

}

inorder(root->right);

}

}

int is\_leaf(TreeNode \*root) { //잎노드인지 판별

return (root->left == NULL && root->right == NULL); //왼쪽 노드와 오른쪽 노드가 없으면 1 아니면 0 반환

}

int eval(TreeNode \*root) {

if (root == NULL) { //입력받은 Node 가 NULL 이면

return 0; //0을 반환한다

}

if (is\_leaf(root)) { //만약 입력받은 노드가 잎노드면

return root->data; //그 노드가 가진 데이터 값 반환

}

int op1 = eval(root->left); //op1 은 왼쪽 노드의 계산 결과

int op2 = eval(root->right); //op2 는 오른쪽 노드의 계산 결과

switch (root->data) //잎노드가 아닌 경우

{

case '+':

return op1 + op2; //부호에 따라 값 결정, 더하기

case '-':

return op1 - op2; //빼기

case '\*':

return op1 \* op2; //곱하기

case '/':

return op1 / op2; //나누기

case '%':

return op1 % op2; //나머지

case '^': //제곱

return pow(op1, op2); //math 헤더의 제곱한 값 반환

}

//그 외의 경우 0반환

return 0;

}

void deleteTree(TreeNode \*root) { //후위 탐색 알고리즘을 이용하여 진행한다

if (root != NULL) { //입력받은 root 가 NULL 이 아니면

deleteTree(root->left); //현 함수를 재귀호출을 한다 (root 의 left에 대해)

deleteTree(root->right); //현 함수를 재귀호출을 한다 (root 의 left에 대해)

free(root); //그 후 마지막으로 방문되는 노드는 root 노드이므로 그 노드를 free 한다.

}

}

void inorder\_iter(TreeNode \*root) {

TreeNode \*curr = root;

Stack s;

Stack\_Ptr stack = &s;

initStack(stack);

while (1) {

while (curr) {

push(stack, (int)curr); //왼쪽으로 이동하면서 모든 노드를 스택에 삽입, 매개변수 타입이 달라서 int 로 형변환

curr = curr->left;

}

if (!is\_empty(\*stack)) {

curr = (TreeNode \*)pop(stack); //스택 반환값이 달라서 TreeNode 형변환

if (is\_leaf(curr)) { //잎노드만 숫자를 가지고 있으므로

printf("[%d]", curr->data); //숫자 출력

}

else { //잎노드가 아닌 경우 부호 출력

printf("[%c]", curr->data);

}

//데이터는 curr 의 오른쪽으로 이동 한다.

curr = curr->right;

}

else {

break;

}

}

printf("\n");

}

void postorder\_iter(TreeNode \*root) {

TreeNode \*curr = root;

Stack s;

Stack\_Ptr stack = &s;

initStack(stack);

TreeNode \*visited = NULL;

while (1) {

if (curr && curr != visited) { //현재 노드가 NULL 이 아니고 방문되지 않은것이면

push(stack, curr); //curr 을 스택에 넣는다.

//후위 탐색은 좌측 우측 루트 이렇게 출력된다.

//스택은 먼저 들어간게 마지막에 나오므로 후위 탐색을 진행할려면

//루트 우측 좌측 노드 이렇게 삽입을 해야한다.

while (curr) { //curr 이 NULL 이 아닐때 까지 반복한다.

//좌측 우측 루트로 나와야 하므로

if (curr->right) { //curr 의 right 가 NULL 이 아니면

push(stack, curr->right); //스택에 넣는다

}//루트는 삽입되었으므로 우측 삽입

if (curr->left) { //curr 의 left 가 NULL 이 아니면

push(stack, curr->left); //스택에 넣는다

}//루트 우측 삽입되었으므로 좌측 삽입

curr = curr->left; //curr 은 좌측 노드로 이동한다.

}

}

if (!is\_empty(\*stack)) { //스택이 비지 않았으면 수행

curr = (TreeNode \*)pop(stack); //스택 반환값이 달라서 TreeNode 형변환

//예외 처리, 왼쪽 노드는 존재하는데 오른쪽 노드는 없는경우

//또한 왼쪽 노드가 방문되지 않은 경우

if (curr->left && curr->right == NULL && curr->left != visited) {

//현재 노드를 삽입하고

push(stack, curr); //루트 우측 좌측인데 우측이 없으므로 루트 좌측

//좌측 노드로 이동한다

curr = curr->left;

}

if (curr->right == NULL || curr->right == visited) {

//오른쪽 노드가 NULL 이거나 오른쪽 노드가 방문된 경우

//후위탐색은 좌측 우측 루트 (본인) 이다.

//위에서까진 좌측 노드가 다 된 경우라 현재는

//우측노드가 완료됬거나 없으면 출력된다

if (is\_leaf(curr)) { //잎노드만 숫자를 가지고 있으므로

printf("[%d]", curr->data); //숫자 출력

}

else { //잎노드가 아닌 경우 문자로 데이터 출력

printf("[%c]", curr->data);

}

//현재 노드를 완료된 노드로 설정

visited = curr;

}

}

else { //스택이 비었으면 while 문 종료

break;

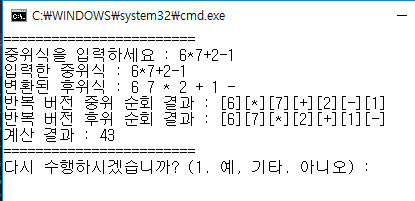
}

}

printf("\n");

}

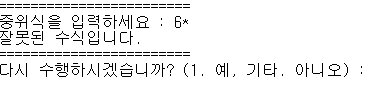
5. 실행결과



1번 경우인 수식이 올바른 경우는, 정상적으로 후위 식 변환 및 평가가 트리를 통해 이루어진다.

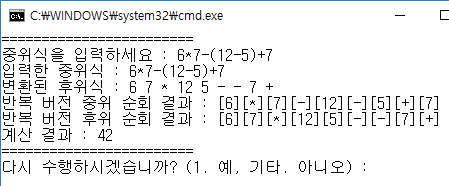
반복버전으로 중위 순회를 하면, 원래 식이 나오고, 후위 순회를 하면, 후위 식이 출력된다

그 후 다시 수행할 것인지 물어본다



2번 경우인 수식이 올바르지 않는 경우는, 식이 올바르지 않다는 에러 메시지를 보여줌을 알 수 있다.

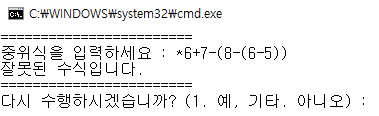
그 후 다시 수행할 것인지 물어본다.



3번 경우인 괄호쌍이 올바르고 수식이 올바른 경우엔, 정상적으로 후위 식 변환 및 평가가 이루어진다.

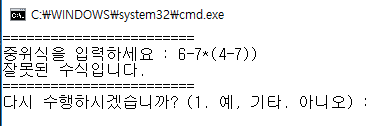
중위 순회를 하면 중위 식이 출력되고, 후위 순회 하면 후위 식이 출력된다. 다만, 중위 순회를 하였을 때 괄호 때문에 연산자 우선순위가 바뀌어야 하지만, 괄호가 없는 식이 출력이 된다.

그 후 다시 수행할 것인지 물어본다



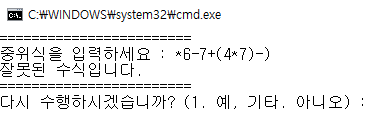
4번 경우인 괄호쌍이 올바르고 수식이 올바르지 않은 경우, 식이 올바르지 않다는 에러가 출력된다.

그 후 다시 수행할 것인지 물어본다



5번 경우인 괄호쌍이 올바르지 않고 수식이 올바른 경우엔, 식이 올바르지 않다는 에러 메시지가 출력된다.

그 후 다시 수행할 것인지 물어본다.



6번 경우인 괄호쌍이 올바르지 않고 수식이 올바르지 않는 경우엔, 식이 올바르지 않다는 에러 메세지가 출력된다.

1 ~ 6 번 경우를 보면서, 올바르지 않았을 때 작동이 되는 경우는 없고, 올바른 경우에서 작동이 안되는 결과는 존재하지 않았다. 즉, 정상적으로 알고리즘이 구현 됬음을 보여준다.

1. 의견

처음 작성했을 때, 반복을 하면서 생성된 루트 노드 및 동적으로 생성된 posted 문자열을 free 처리 하지 않았습니다. 그렇기 때문에 메모리 누수가 일어났습니다. 프로그램을 다 짜고 제출을 하기 전에, 뭔가 이상해서 봤더니 동적 할당의 free 처리를 하지 않은 것을 뒤늦게 알아차렸습니다.

제출하기 직전에 다시 또, 동적 할당의 free 를 생각해봤 어야 했습니다. 제일 까다로웠던 것은, 루트 노드의 초기화 였습니다. 반복을 사용하여 구현을 하려다, 스택을 사용하고 기타 작업을 더 해야 돼서, 순환을 가지고 하였습니다. 순환을 가지고 동적 할당을 해제 할 때, 후위 탐색 알고리즘을 작성했는데, 그 이유는 루트가 먼저 사라지면 루트에 연결되어있는 하위 노드들은 참조 할 수 없기 때문이었습니다.

마지막으로, 변환된 후위 식을 출력하는 부분에서 문제가 생겼었습니다.

루트 노드를 만드는 과정에서 strtok를 사용하여 매개변수로 받아온 스트링을 공백을 기준으로 잘랐습니다. Strtok 를 사용하면, 원본포인터가 바뀌는 것을 까먹고 strtok 를 진행 하고 즉, 루트를 생성하는 함수를 호출하고 후위 식을 출력하였었습니다. 이 문제를 나중에 알아차려서, 후위 식을 출력하고 후위 식으로 루트 노드를 만드는 것으로 알고리즘을 해결했습니다.

여담으로 제작한 소스 전체 코드는

<https://github.com/wookoo/Data-Structure/blob/master/7.%20%ED%8A%B8%EB%A6%AC/5.%20%ED%8A%B8%EB%A6%AC%20-%20%EC%A4%91%EC%9C%84%EC%8B%9D%EC%9D%84%20%ED%8A%B8%EB%A6%AC%EB%A1%9C%20%EA%B5%AC%EC%84%B1.c> 에서 확인 하실 수 있습니다.

아래의 QR 을 스캔하시면 위의 링크로 이동됩니다.



<< 원본 소스로 이동하는 QR코드