

1. 과제 개요

일반적으로 연산자와 피연산자 그리고 괄호로 이루어진 수식은 전위식(prefix), 중위식(infix), 후위식(postfix)으로 나뉜다. 이번 과제1(스택계산기)의 경우 중위식이 입력되었을 때, 이를 각각 전위식과 후위식으로 변환하고, 추가로 후위식의 경우 연산을 하여 이에 대한 결과를 도출해내는 것이 주요 요구사항이고, 부가적으로 소수점처리와 문자열입력에 대한 오류처리를 하는 요구사항이 존재한다. 이러한 결과를 도출해내기 위해서는 링크드 리스트(linked list)로 각각의 노드와 스택의 로직을 설계하여 구현해야하고, 부가적인 요구사항들을 만족시키는 프로그램을 작성해야한다.

2. 설계

스택계산기 프로그램은 크게 1.검사 2.변환 3.계산의 3가지 로직이 요구된다.

먼저 첫번째의 검사의 경우, 입력된 중위식에서 괄호의 쌍이 맞는지를 검증해야하고, 괄호의 순서가 서로 꼬이지 않았는지 검증도 요구되었다. 이러한 괄호검사의 경우, stack을 활용하여 검사를 진행한다. Stack을 생성하고 입력된 중위식 문자열을 순회하며, 숫자와 소수점을 제외한 괄호를 만날 때마다, 여는 괄호의 경우 괄호를 stack에 push하고, 닫는 괄호의 경우, stack에서 pop하여 닫는괄호와 동일한 괄호가 사용여부를 검증하며 꼬인 괄호의 검출을 한다. 이렇게 모든 문자열의 순회이후, stack에 존재하는 괄호를 확인하여 만약 stack에 괄호가 존재하는 경우, 괄호의 짝이 안 맞는 오류를 검출해낼 수 있다.

또한, 문자열의 입력여부는 괄호검사과정 중, 괄호를 제외한 숫자와 소수점을 검사하는 로직에서 두가지 경우를 제외한 문자가 검출되었을 때, 이를 문자의 입력으로 판단하여 오류검출을 진행한다.

마지막으로 부호검증의 경우, 중위식의 전위식 변환과 중위식의 후위식 변환 과정에서 중위식의 문자열을 순회하면서 부호(+, -, *, /)가 연속하여 존재하는 경우, 이를 부호에 대한 오류로 판단하는 로직을 구현하였다.

두번째 중심 요구사항인 변환의 경우, 크게 중위식을 전위식으로 변환하는 함수와 중위식을 후위식으로 변환하는 함수를 작성하였다. 중위식을 전위식으로 변환하는 로직은 lined stack type 구조체를 활용하여 임의의 stack을 생성한 뒤, 입력된 중위식문자열을 첫문자부터 순회하며 모두 stack에 push한 뒤, 이를 모두 pop하여 중위식문자열을 반전시킨다. Stack으로부터 pop된 문자열을 활용하여 전위식 변환을 진행하는데, 이를 위해서는 연산자에 대한 stack과 데이터(숫자)에 대한 stack을 선언하여 변환을 진행했다. 역순으로 저장된 중위식문자열을 stack으로부터 pop하여

숫자값이 나오는 경우, 이를 바로 데이터 stack에 push하고, 연산자값이 나오는 경우에는 부호검증을 통해 pop된 이후의 stack에 부호값이 또 들어있는지 확인한 후, 문제가 없다면 연산자 stack의 top에 저장된 값과 연산자의 우선순위를 반환해주는 prec함수를 통해 우선순위를 비교한다. 만약, 연산자 stack의 top에 저장된 값이 우선순위가 더 높은 경우 top에 저장된 값을 pop하여 데이터 stack에 push 하고, 우선순위가 더 작은 연산자가 확인되거나 연산자 stack이 비어있는 경우 순회를 빠져나온다. 이러한 과정 뒤에는 비교에 사용했던 연산자를 연산자 stack에 push한다. 연산자를 제외한 괄호 같은 경우, 닫는 괄호는 연산자 stack에 push하고, 여는 괄호가 나오는 경우, 닫는 괄호가 나올 때까지 모든 연산자를 데이터 stack에 옮겨서 push 시킨다. 이 모든 과정 이후, 만약 연산자 stack에 값이 존재하면 stack이 빌 때까지 연산자들을 데이터 stack에 옮긴다. 모든 데이터가 전위식으로 바뀌어 stack에 순서대로 저장되었기 때문에, 데이터 stack의 모든 값을 pop하여 빈 문자열에 하나씩 추가시킨다. 소수점 값의 경우 다음값으로 숫자가 아닌값이 나올 때까지 모든 값을 공백문자 없이 다 붙여서 추가시킨다. 데이터 stack에 저장된 모든 값의 추가가 완료되면, 추가가 완료된 전위식 문자열을 return 한다.

중위식을 후위식으로 바꾸는 로직의 경우, 우선 연산자의 우선순위를 파악하기 위해 연산자 stack을 선언했다. 그리고 인자로 들어온 문자열을 순회하면서 숫자인 경우에는 소수점이 없는 경우, 값을 바로 문자열에 추가시키고, 소수점을 확인하여, 소수점이 존재하는 경우에는 연산자와 괄호를 제외한 모든 값을 문자열 뒤에 추가시켜서 하나의 항으로 인식될 수 있도록 로직을 설계했고, 연산자가 입력되는 경우에는 연산자의 우선순위를 비교하여 연산자 stack의 top에 존재하는 연산자의 우선순위가 더 높은 경우에는 낮은 연산자가 확인될 때까지 모든 값을 문자열에 pop하여 추가시킨다. 이러한 과정 이후에는 입력된 연산자를 stack에 push 한다. 여는 괄호일 경우에는 stack으로 push하고, 닫는 괄호일 경우에는 여는 괄호가 나올 때까지 문자열에 stack의 값을 pop하여 추가시킨다. 이 과정이 끝나면 stack에 남은 값을 문자열에 추가시킨 뒤, 문자열을 return 한다.

마지막 중심 요구사항인 후위식 연산의 경우 먼저 연산을 위한 stack을 선언하고, 각각의 피연산자를 사용하기 위한 변수2개, 연산결과 값을 담기위한 result 변수를 선언해야한다. 입력받은 문자열을 순회하면서 값이 숫자인 경우, 뒤의 값을 확인하여 소수점이 존재하는 경우, 이를 반영하는 로직을 설계해야한다. 소수점의 자릿수만큼 값에 0.1을 곱하여 소수점 위치를 설정한다. 숫자 하나를 완전히 얻은 이후에는 이 값을 stack에 push한다. 값이 연산자인 경우에는 stack의 top값을 push하여 오른쪽 피연산자로, 그 다음 값을 push하여 왼쪽 피연산자로 사용해야 한다. switch문을 활용하여 연산자에 맞춰 연산을 진행한 뒤에 값을 다시 stack에 push한다. 모든 문자열을 순회한 뒤에는 stack에 저장된 결과값을 return 한다.

3. 구현

- 각 함수별 기능

프로그램에 사용된 각함수의 기능은 다음과 같이 정리할 수 있다.

1) void init(LinkedListType* s)

인 자 : LinkedListType pointer

리턴형 : void

설 명 : Stack의 top을 null로 할당하여, LinkedListType을 초기화 한다.

2) element isEmpty(LinkedListType* s)

인 자 : LinkedListType pointer

리턴형 : element(int)

설 명 : stack이 비어있는지 확인하는 함수로써, top값과 NULL값의 비교식을 return 한다.

3) element pop(LinkedListType* s)

인 자 : LinkedListType pointer

리턴형 : element(int)

설 명 : stack의 top에 위치하고 있는 값을 return하고, stack에 값이 없을 경우 error 메시지를 출력한다.

4) double fpop(LinkedListType* s)

인 자 : LinkedListType pointer

리턴형 : double

설 명 : 실수값의 저장을 위해 만들어진 함수로, type을 제외한 나머지 로직은 pop함수와 동일하다.

5) void push(LinkedStackType* s, element item)

인 자 : 1) LinkedStackType pointer 2) item(element)

리턴형 : void

설 명 : stack에 값을 저장하기 위해 고안된 함수로써, StackNode pointer type의 temp로 메모리할당을 받아, node에 data를 저장하고, stack의 top에 node의 주소를 할당시킨다.

6) void fpush(LinkedStackType* s, double item)

인 자 : 1) LinkedStackType pointer 2) item(double)

리턴형 : void

설 명 : double형 data를 저장하기 위해 고안된 함수로써, push와 동일한 로직으로 동작한다.

7) element peek(LinkedStackType* s)

인 자 : LinkedStackType pointer

리턴형 : element(int)

설 명 : stack의 top에 위치한 data를 return하는 함수로써, stack이 비어있으면, NULL을 return 한다.

8) double fpeek(LinkedStackType* s)

인 자 : LinkedStackType

리턴형 : double

설 명 : stack의 top에 위치한 fdata를 return하는 함수로써, stack이 비어있으면, NULL을 return 한다.

9) int isFull(LinkedStackType* s)

인 자 : LinkedStackType pointer

리턴형 : int

설 명 : stack이 가득찼는지 여부를 return 한다, 하지만, 이론상 full상태는 발생하지 않으므로, 0을 반환

10) int bracketCheck(char* ch)

인 자 : char pointer ch

리턴형 : int

설 명 : 주어진 문자열에서 괄호검사를 진행하여 문제가 없을시 1 return, 문제발생시 0 return

11) int prec(char op)

인 자 : char op

리턴형 : int

설 명 : 연산자의 우선순위를 반환하기 위한 함수

12) char* infix_to_postfix(char* ch)

인 자 : char pointer ch

리턴형 : char pointer

설 명 : 중위식 문자열을 입력받아서 후위식 문자열을 return 하는 함수

13) char* infix_to_prefix(char* ch)

인 자 : char pointer ch

리턴형 : char pointer

설 명 : 중위식 문자열을 전위식 문자열로 바꿔주는 함수

14) double postfix_calculate(char* ch)

인 자 : char pointer ch

리턴형 : double

설 명 : 후위식 연산결과를 return 하기 위한 함수

15) int main(void)

인 자 : void

리턴형 : int

설 명 : main 함수, 실행시 최초로 실행되는 함수

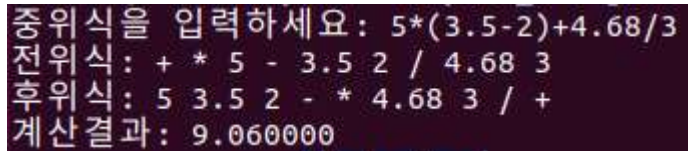
4. 테스트 및 결과

- 테스트 프로그램의 실행 결과를 분석 (캡처화면과 함께)

모든 합격/불합격 케이스 입력

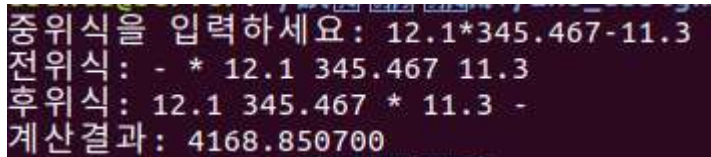
- 합격 케이스

- 1) 명세서에 제시된 예시 $5*(3.5-2)+4.68/3$



중위식을 입력하세요: $5*(3.5-2)+4.68/3$
전위식: + * 5 - 3.5 2 / 4.68 3
후위식: 5 3.5 2 - * 4.68 3 / +
계산결과: 9.060000

- 2) 두 자리수 이상의 입력 $12.1*345.467-11.3$



중위식을 입력하세요: $12.1*345.467-11.3$
전위식: - * 12.1 345.467 11.3
후위식: 12.1 345.467 * 11.3 -
계산결과: 4168.850700

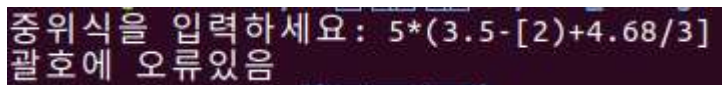
- 불합격 케이스

- 1) 괄호의 쌍이 맞지 않는 경우 에러발생 $5*(3.5-2)+4.68/3)$



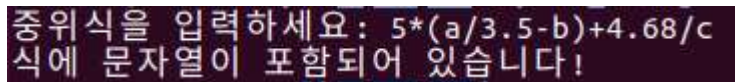
중위식을 입력하세요: $5*(3.5-2)+4.68/3)$
괄호에 오류있음

- 2) 괄호가 서로 뒤바뀐 경우 에러발생 $5*(3.5-[2])+4.68/3]$



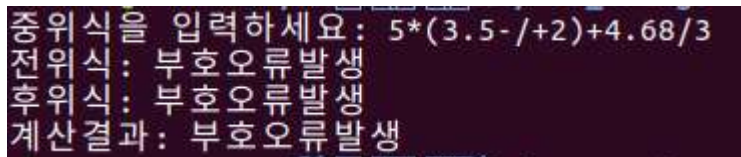
중위식을 입력하세요: $5*(3.5-[2])+4.68/3]$
괄호에 오류있음

- 3) 문자열이 입력된 경우 에러발생 $5*(a/3.5-b)+4.68/c$



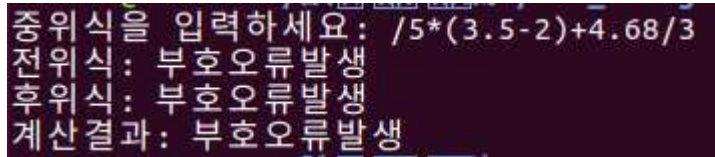
중위식을 입력하세요: $5*(a/3.5-b)+4.68/c$
식에 문자열이 포함되어 있습니다!

- 4) 부호가 연속으로 입력된 경우 에러발생 $5*(3.5-/+2)+4.68/3$



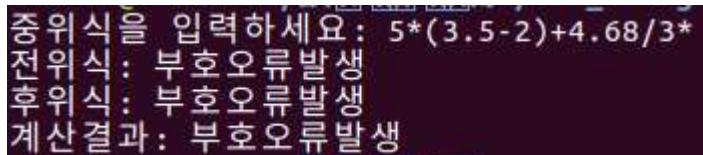
중위식을 입력하세요: $5*(3.5-/+2)+4.68/3$
전위식: 부호오류발생
후위식: 부호오류발생
계산결과: 부호오류발생

5) 부호가 식의 맨 앞에 입력된 경우 에러발생 $/5*(3.5-2)+4.68/3$



중위식을 입력하세요: $/5*(3.5-2)+4.68/3$
전위식: 부호오류발생
후위식: 부호오류발생
계산결과: 부호오류발생

6) 부호가 식의 맨 뒤에 입력된 경우 에러발생 $5*(3.5-2)+4.68/3*$



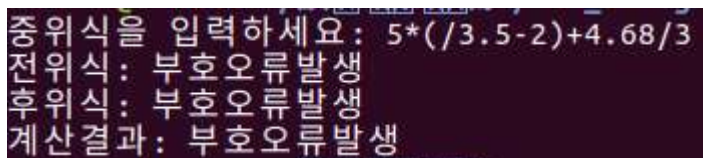
중위식을 입력하세요: $5*(3.5-2)+4.68/3*$
전위식: 부호오류발생
후위식: 부호오류발생
계산결과: 부호오류발생

7) 부호가 닫는 괄호의 앞에 입력된 경우 에러발생 $5*(3.5-2-)+4.68/3$



중위식을 입력하세요: $5*(3.5-2-)+4.68/3$
전위식: 부호오류발생
후위식: 부호오류발생
계산결과: 부호오류발생

8) 부호가 여는 괄호의 뒤에 입력된 경우 에러발생 $5*(/3.5-2)+4.68/3$



중위식을 입력하세요: $5*(/3.5-2)+4.68/3$
전위식: 부호오류발생
후위식: 부호오류발생
계산결과: 부호오류발생

5. 소스코드 (주석포함)

```
// 202101544 마영준
```

```
// stack calculator
```

```
// 헤더파일 작성
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
// int type을 element라는 이름으로 사용
```

```
typedef int element;
```

```
// stack 사용을 위한 node 구조체선언
```

```
typedef struct StackNode {
```

```
    element data;
```

```
    double fdata;
```

```
    // 자기참조구조체 선언
```

```
    struct StackNode* link;
```

```
} StackNode;
```

```
// node의 linked list로써의 사용을 위한 linkedstacktype 구조체선언
```

```
typedef struct {
```

```
    StackNode* top;
```

```
} LinkedStackType;
```

```
/*
```

```
    함수명 : init
```

인 자 : LinkedStackType pointer

리턴형 : void

설 명 : Stack의 top을 null로 할당하여, LinkedStackType을 초기화 한다.

*/

```
void init(LinkedStackType* s) {
```

```
    s->top = NULL;
```

```
}
```

/*

함수명 : isEmpty

인 자 : LinkedStackType pointer

리턴형 : element(int)

설 명 : stack이 비어있는지 확인하는 함수로써, top값과 NULL값의 비교식을 return 한다.

*/

```
element isEmpty(LinkedStackType* s) {
```

```
    return (s->top == NULL);
```

```
}
```

/*

함수명 : pop

인 자 : LinkedStackType pointer

리턴형 : element(int)

설 명 : stack의 top에 위치하고 있는 값을 return하고, stack에 값이 없을 경우 error 메시지를 출력한다.

*/

```
element pop(LinkedStackType* s) {
```

```

    if (isEmpty(s)) {

        fprintf(stderr, "스택이 비어있음\n");

        exit(1);

    }

    else {

        StackNode* temp = s->top;

        int data = temp->data;

        s->top = s->top->link;

        free(temp);

        return data;

    }

}

```

/*

함수명 : fpop

인 자 : LinkedStackType pointer

리턴형 : double

설 명 : 실수값의 저장을 위해 만들어진 함수로, type을 제외한 나머지 로직은 pop함수와 동일하다.

*/

```

double fpop(LinkedStackType* s) {

    if (isEmpty(s)) {

        fprintf(stderr, "스택이 비어있음\n");

        exit(1);

    }

    else {

        StackNode* temp = s->top;

```

```

        double fdata = temp->fdata;

        s->top = s->top->link;

        free(temp);

        return fdata;

    }

}

```

/*

함수명 : push

인 자 : 1) LinkedStackType pointer 2) item(element)

리턴형 : void

설 명 : stack에 값을 저장하기 위해 고안된 함수로써, StackNode pointer type의 temp로 메모리할당을 받아, node에 data를 저장하고, stack의 top에 node의 주소를 할당시킨다.

*/

```
void push(LinkedStackType* s, element item) {
```

```
    StackNode* temp = (StackNode*)malloc(sizeof(StackNode));
```

```
    temp->data = item;
```

```
    temp->link = s->top;
```

```
    s->top = temp;
```

```
}
```

/*

함수명 : fpush

인 자 : 1) LinkedStackType pointer 2) item(double)

리턴형 : void

설 명 : double형 data를 저장하기 위해 고안된 함수로써, push와 동일한 로직으로 동작한다.

*/

```
void fpush(LinkedStackType* s, double item) {
```

```
    StackNode* temp = (StackNode*)malloc(sizeof(StackNode));
```

```
    temp->data = item;
```

```
    temp->link = s->top;
```

```
    s->top = temp;
```

```
}
```

```
/*
```

```
    함수명 : peek
```

```
    인 자 : LinkedStackType pointer
```

```
    리턴형 : element(int)
```

```
    설 명 : stack의 top에 위치한 data를 return하는 함수로써, stack이 비어있으면, NULL을  
    return 한다.
```

```
*/
```

```
element peek(LinkedStackType* s) {
```

```
    if (isEmpty(s)) {
```

```
        // stack이 비어있는 경우 NULL return
```

```
        return NULL;
```

```
    }
```

```
    else {
```

```
        // stack의 top에 위치한 data return
```

```
        return s->top->data;
```

```
    }
```

```
}
```

```
/*
```

```
    함수명 : fpeek
```

인 자 : LinkedStackType

리턴형 : double

설 명 : stack의 top에 위치한 fdata를 return하는 함수로써, stack이 비어있으면, NULL을 return 한다.

*/

```
double fpeek(LinkedStackType* s) {
```

```
    if (isEmpty(s)) {
```

```
        // stack이 비어있는 경우 NULL return
```

```
        return 0;
```

```
    }
```

```
    else {
```

```
        // stack의 top에 위치한 data return
```

```
        return s->top->fdata;
```

```
    }
```

```
}
```

/*

함수명 : isFull

인 자 : LinkedStackType pointer

리턴형 : int

설 명 : stack이 가득찼는지 여부를 return 한다, 하지만, 이론상 full상태는 발생하지 않으므로, 0을 반환

*/

```
int isFull(LinkedStackType* s) {
```

```
    // linkedlist에서 이론상으로 full상태는 발생하지 않음 (다른 메모리영역을 침범하는 경우 제외)
```

```
    return 0;
```

```
}
```

/*

함수명 : bracketCheck

인 자 : char pointer ch

리턴형 : int

설 명 : 주어진 문자열에서 괄호검사를 진행하여 문제가 없을시 1 return, 문제발생시 0
return

*/

int bracketCheck(char* ch) {

// stack 선언

LinkedListType s;

// stack 초기화

init(&s);

// 문자열 순회를 위한 index 선언

int i = 0;

// 주어진 문자열을 순회하며 괄호검사를 진행한다

// 문자열의 마지막값은 NULL이므로, NULL이 아닌 이상 순회를 지속한다.

while (ch[i] != NULL) {

// ch 문자열의 각 값을 활용하여 분기별로 판단하기 위해 switch문을 사용

switch (ch[i])

{

// 여는 괄호인 경우, 괄호를 그대로 stack에 push한다.

case '(': case '[': case '{':

push(&s, ch[i]);

break;

// 닫는 괄호인 경우, 추가적인 로직을 사용하여 검사를 진행한다.

case ')': case ']': case '}':

// 닫는 괄호를 만났을 때 stack이 비어있는 경우, 이는 닫는 괄호
한개가 남는 상황이므로 실패인 0을 return 한다.

```
if (isEmpty(&s)) {
```

```
    return 0;
```

```
}
```

// stack에 값이 존재하는 경우 이 값을 pop하여 검사를 진행한다.

```
else {
```

```
    // pop한 값 data변수에 할당
```

```
    element data = pop(&s);
```

```
    // switch문을 사용하여, stack의 값을 판단
```

```
    switch (data)
```

```
    {
```

```
    // 동일한 짝의 괄호가 아닌 경우 실패인 0을 return
```

```
    case '(':
```

```
        if (ch[i] != ')') {
```

```
            return 0;
```

```
        }
```

```
        break;
```

```
    case '[':
```

```
        if (ch[i] != ']') {
```

```
            return 0;
```

```
        }
```

```
        break;
```

```
    case '{':
```

```
        if (ch[i] != '}') {
```

```
            return 0;
```

```
        }
```



```

        }

    }

    // 동일한 괄호인 경우 별도의 할당없이 break

    break;

default:

    // 문자입력 검출, ASCII TABLE에서 문자입력에 해당하는 65~90, 97~122
구간의 값이 입력된 경우, 오류검출

    if ((ch[i]>=65 && ch[i] <= 90) || (ch[i]>=97 && ch[i]<=122)) {

        printf("식에 문자열이 포함되어 있습니다!\n");

        exit(1);

    }

    break;

}

// index를 증가시키며 ch문자열을 순회한다.

i++;

}

// 괄호 검사 후, stack에 남아있는 data가 없는 경우 성공인 1을 return

if (isEmpty(&s)) {

    return 1;

}

// 괄호 검사 후, stack에 값이 남아있는 경우, 괄호의 짝이 맞지않는 것이므로 실패인
0을 return

else {

    return 0;

}

}

/*

```

함수명 : prec

인 자 : char op

리턴형 : int

설 명 : 연산자의 우선순위를 반환하기 위한 함수

*/

```
int prec(char op) {  
    switch (op) {  
        // 괄호의 경우, 0의 우선순위를 반환  
        case '(': case ')': return 0;  
        // +-의 경우, 1의 우선순위를 반환  
        case '+': case '-': return 1;  
        // */의 경우, 2의 우선순위를 반환  
        case '*': case '/': return 2;  
    }  
    // 이외 연산자의 경우 -1의 우선순위를 반환  
    return -1;  
}
```

/*

함수명 : infix_to_postfix

인 자 : char pointer ch

리턴형 : char pointer

설 명 : 중위식 문자열을 입력받아서 후위식 문자열을 return 하는 함수

*/

```
char* infix_to_postfix(char* ch) {  
    // 후위식 문자열을 저장한 char pointer를 선언함  
    char* str = (char*)malloc(sizeof(char) * 100);
```

```

// strcpy를 활용하여 str 초기화

strcpy(str, "");

// 중위식의 후위식 변환을 위한 stack선언

LinkedStackType* s;

// stack 초기화

init(&s);

// 중위식 값을 받을 element(int) type 변수 선언

element e;

// 중위식 문자열의 순회를 위한 index 변수 선언

int i = 0;

// 시작값이 부호인경우, 부호오류발생으로 판단, -1 return

if (ch[i] == '+' || ch[i] == '-' || ch[i] == '*' || ch[i] == '/') {

    strcpy(str, "부호오류발생");

    return -1;

}

// 중위식 문자열을 순회하는데, 문자열은 마지막 문자가 NULL이므로, 값이 NULL이
아닌동안 순회를 한다.

while (ch[i] != NULL) {

    // index에 따라 중위식 문자열의 값을 switch문을 활용하여 판단

    switch (ch[i])

    {

        // ch[i]가 연산자인 경우

        case '+': case '-': case '/': case '*':

            // 연산자 이후에 연산자가 오거나 닫는괄호가 오는 경우,
            부호오류발생으로 판단, -1 return

            if (ch[i + 1] == '+' || ch[i + 1] == '-' || ch[i + 1] == '/' || ch[i + 1] == '*'

                || ch[i + 1] == ')' || ch[i + 1] == '}' || ch[i + 1] == ']') {

                strcpy(str, "부호오류발생");

```

```

        return -1;
    }

    // stack이 비어있는지 여부를 확인하고, 연산자의 우선순위를
    prec함수를 통해 비교하여 peek을 통해 확인한 stack의 top에 존재하는 연산자의 우선순위가 큰
    경우, stack에 존재하는 연산자를 pop하여, 후위식문자열에 추가한다.

    while (!isEmpty(&s) && (prec(ch[i]) <= prec(peek(&s)))) {

        // stack의 top에 위치한 data pop

        e = pop(&s);

        // 후위식 문자열에 추가

        strncat(str, &e, 1);

        // 추후 구분을 위해 공백도 추가

        strcat(str, " ");

    }

    // ch[i]는 stack으로 push

    push(&s, ch[i]);

    break;

// 여는 괄호일 경우, stack으로 push

case '(': case '[': case '{':

    // 여는 괄호 바로 뒤에 연산자가 오는 경우 부호오류발생으로 판단, -1

return

    if (ch[i + 1] == '+' || ch[i + 1] == '-' || ch[i + 1] == '*' || ch[i + 1] == '/')

{

        strcpy(str, "부호오류발생");

        return -1;

    }

    push(&s, ch[i]);

    break;

// 닫는 괄호일 경우, stack에서 값이 여는 괄호가 나올때까지 pop을 반복한다.

```

```

case ')': case ']': case '}':

    e = pop(&s);

    // 여는 괄호가 아닌동안에는 pop한 값을 모두 후위식문자열에
    붙여넣는다.

    while (e != '(' && e != '[' && e != '{') {

        strncat(str, &e, 1);

        strcat(str, " ");

        e = pop(&s);

    }

    break;

// 연산자와 괄호가 아닌경우의 값은 default로직을 통해 처리한다.

default:

    if (ch[i+1] >= '0' && ch[i+1] <= '9') {

        strncat(str, &ch[i], 1);

        while (ch[i + 1] != NULL && ch[i + 1] >= 48 && ch[i + 1] <=

57) {

            i++;

            strncat(str, &ch[i], 1);

        }

        if (ch[i + 1] != NULL && ch[i + 1] == '.') {

        }

        else {

            // 띄어쓰기 추가

            strcat(str, " ");

        }

    }

    // 만약 data의 뒤에 소수점의 존재를 의미하는 '.'가 있는 경우,

```

```

else if (ch[i + 1] == '.') {

    // data를 후위식 문자열에 추가

    strncat(str, &ch[i], 1);

    // index를 '.' 위치로 설정

    i++;

    // '.'도 후위식 문자열에 추가

    strncat(str, &ch[i], 1);

    // data의 다음 값이 NULL 또는 숫자 이외의 값이 아닌 이상,
뒤에 오는 모든 값을 소수점 아랫자리라고 생각하고 문자열에 추가

    while (ch[i+1] != NULL && ch[i+1] >= 48 && ch[i+1] <= 57) {

        // index 증가

        i++;

        // 문자열에 ch[i] 값 추가

        strncat(str, &ch[i], 1);

    }

    // 띄어쓰기 추가

    strcat(str, " ");

}

else {

    // 일반적인 숫자의 경우, 문자열에 추가

    strncat(str, &ch[i], 1);

    // 띄어쓰기 추가

    strcat(str, " ");

}

break;

}

```

```

        // index 증가

        i++;

    }

    // 마지막값이 부호인경우, 부호오류발생으로 판단, -1 return
    if (ch[i-1] == '+' || ch[i-1] == '-' || ch[i-1] == '*' || ch[i-1] == '/') {

        strcpy(str, "부호오류발생");

        return -1;

    }

    // stack에 값이 남아있는 경우, 나머지 값을 다 반환하여 문자열에 추가
    while (!isEmpty(&s)) {

        e = pop(&s);

        // stack에서 pop한 뒤 stack에 data가 남아있는 경우 띄어쓰기 추가
        if (!isEmpty(&s)) {

            strncat(str, &e, 1);

            strcat(str, " ");

        }

        // stack에 값이 없는 경우에는 띄어쓰기를 추가하지 않는다.
        else {

            strncat(str, &e, 1);

        }

    }

    // 후위식 문자열 반환

    return str;

}

/*

```

함수명 : infix_to_prefix

인 자 : char pointer ch

리턴형 : char pointer

설 명 : 중위식 문자열을 전위식 문자열로 바뀌주는 함수

*/

```
char* infix_to_prefix(char* ch) {  
    // 문자열의 역순정렬을 위한 stack 선언  
    LinkedStackType* s;  
    // 연산자를 저장하기 위한 stack 선언  
    LinkedStackType* op;  
    // data를 저장하기 위한 stack 선언  
    LinkedStackType* dt;  
    // stack 초기화  
    init(&s);  
    init(&op);  
    init(&dt);  
    // 전위식 문자열을 저장하기 위한 char pointer 선언  
    char* pre_str = (char*)malloc(sizeof(char) * 100);  
    // 전위식 문자열 초기화  
    strcpy(pre_str, "");  
    // data와 연산자를 다루기 위한 변수 선언  
    element data, oper;  
    // index 변수선언  
    int i = 0;  
    // 시작값이 부호인경우, 부호오류발생으로 판단, -1 return  
    if (ch[i] == '+' || ch[i] == '-' || ch[i] == '*' || ch[i] == '/') {  
        strcpy(pre_str, "부호오류발생");  
    }  
}
```



```

        return -1;

    }

    // 중위식 문자열의 값을 역순으로 뒤집기 위해 만든 순회, 문자열의 마지막인 NULL을
    만날 때까지 순회한다.

    while (ch[i] != NULL) {

        if ((ch[i] >= '0' && ch[i] <= '9') || ch[i] == '.') {

            // 두자리수 이상의 숫자일 경우 처리

            if (ch[i + 1] >= '0' && ch[i + 1] <= '9') {

                push(&s, ch[i]);

                i++;

                push(&s, ch[i]);

                while (ch[i + 1] != NULL && ch[i + 1] >= '0' && ch[i + 1] <=
'9') {

                    i++;

                    push(&s, ch[i]);

                }

                if (ch[i+1] != NULL && ch[i+1] == '.') {

                }

                else {

                    push(&s, NULL);

                }

            }

            // data의 다음값이 소수점을 나타내는 '.'인 경우, 별도의 처리를 한다

            else if (ch[i + 1] == '.') {

                // 현재 data를 stack에 push

                push(&s, ch[i]);

                // index를 '.'위치로 변경

```

```

        i++;

        // '.'을 push
        push(&s, ch[i]);

        // data의 다음값이 NULL이 아니면서 숫자일 경우, 값 push
        while (ch[i + 1] != NULL && ch[i + 1] >= 48 && ch[i + 1] <=

57) {

            // index 증가

            i++;

            // data push

            push(&s, ch[i]);

        }

        // 항 1개의 data 저장이 끝난후, NULL 값 push하여 구분

        push(&s, NULL);

    }

    else {

        // data가 '.'이 아닌경우, data push

        push(&s, ch[i]);

    }

}

else {

    // data가 '.'이 아닌경우, data push

    push(&s, ch[i]);

}

// index 증가

i++;

}

// 마지막값이 부호인경우, 부호오류발생으로 판단, -1 return

```

```

if (ch[i-1] == '+' || ch[i-1] == '-' || ch[i-1] == '*' || ch[i-1] == '/') {

    strcpy(pre_str, "부호오류발생");

    return -1;

}

// stack이 빌때까지 반복

while (!isEmpty(&s)) {

    // s stack의 top에서 data pop

    data = pop(&s);

    // data를 가지고 switch를 활용하여 전위식으로의 변환

    switch (data)

    {

        // 연산자/괄호인경우

        case '+': case '-': case '/': case '*':

            // 연산자 이후에 연산자가 오거나 여는괄호가 오는 경우,
            부호오류발생으로 판단, -1 return

            if (peek(&s) == '+' || peek(&s) == '-' || peek(&s) == '/' || peek(&s) == '*'

                || peek(&s) == '(' || peek(&s) == '{' || peek(&s) == '[') {

                strcpy(pre_str, "부호오류발생");

                return -1;

            }

        // 연산자 스택의 최상위 값과 우선순위 비교

        while (!isEmpty(&op) && (prec(data) <= prec(peek(&op)))) {

            // 연산자 stack의 top에 위치한 data의 우선순위가 큰 경우

            stack에서 data pop

            oper = pop(&op);

            // 데이터 stack으로 연산자 push

            push(&dt, oper);

        }

```

```

// data를 연산자 stack으로 push
push(&op, data);

break;

case ')': case ']': case '}':

    // 닫는 괄호 이후에 연산자가 오는 경우, 부호오류발생으로 판단, -1
return

if (peek(&s) == '+' || peek(&s) == '-' || peek(&s) == '*' || peek(&s) == '/')

{

    strcpy(pre_str, "부호오류발생");

    return -1;

}

// 닫는 괄호인 경우, 연산자 stack에 data push
push(&op, data);

break;

case '(': case '[': case '{':

    // 여는 괄호인 경우, 연산자 stack에서 data pop
oper = pop(&op);

// 연산자 data를 데이터 stack에 push
push(&dt, oper);

// 닫는 괄호가 나올때까지 연산자 stack에서 값 pop하여 데이터
stack에 값 push

while (oper != ')' && oper != ']' && oper != '}') {

    oper = pop(&op);

    push(&dt, oper);

}

break;

// 연산자와 괄호를 제외한 경우, 데이터 stack에 값 push

```

```

        default:

            push(&dt, data);

            break;

        }

    }

    // 연산자 stack의 값을 모두 pop하여 데이터 stack에 push
    while (!isEmpty(&op)) {

        oper = pop(&op);

        push(&dt, oper);

    }

    // 데이터 stack이 빌때까지 반복
    while (!isEmpty(&dt)) {

        // 데이터 stack의 top의 값 pop

        data = pop(&dt);

        // 괄호의 경우 생략

        if (data != '(' && data != '[' && data != '{' && data != ')' && data != ']' &&
data != '}') {

            if (data != '+' && data != '-' && data != '*' && data != '/' && peek(&dt)
>= '0' && peek(&dt) <= '9') {

                strcat(pre_str, &data, 1);

                while (!isEmpty(&dt) && peek(&dt) >= 48 && peek(&dt) <= 57)

{

                    data = pop(&dt);

                    strcat(pre_str, &data, 1);

                }

                if (!isEmpty(&dt) && peek(&dt) == '.') {

                    }

                }

            }

        }

    }

```

```

        else {

            strcat(pre_str, " ");

        }

    }

    // 데이터 stack의 top값이 소수점인 경우
    else if (peek(&dt) == '.') {

        // 전위식에 data 추가

        strncat(pre_str, &data, 1);

        // '.' pop으로 data에 할당

        data = pop(&dt);

        // 전위식에 소수점 추가

        strncat(pre_str, &data, 1);

        // 숫자 스택의 peek값이 숫자인 경우, space없이 pre_str에

        while (!isEmpty(&dt) && peek(&dt) >= '0' && peek(&dt) <= '9')

        {

            data = pop(&dt);

            strncat(pre_str, &data, 1);

        }

        strcat(pre_str, " ");

    }

    else {

        // 데이터 stack의 top값이 소수점이 아닌 경우

        strncat(pre_str, &data, 1);

        strcat(pre_str, " ");

    }

}

}

```

추가

```

// 전위식 문자열 return

return pre_str;

}

/*

함수명 : postfix_calculate

인 자 : char pointer ch

리턴형 : double

설 명 : 후위식 연산결과를 return 하기 위한 함수

*/

double postfix_calculate(char* ch) {

    // 후위식 연산을 위한 stack 선언

    LinkedStackType* s;

    // stack 초기화

    init(&s);

    // 피연산자를 저장하기 위한 변수 op1, op2, 계산결과를 저장하기 위한 value 선언

    double op1, op2, value;

    // 문자열 순회를 위해 index 선언

    int i = 0;

    // 소수점 처리를 위한 decimal 선언

    int decimal;

    // 소수점을 저장하기 위한 d_value 선언

    double d_value;

    // 문자열 전체를 순회

    while (ch[i] != NULL) {

        // 항목이 피연산자일 경우

        if (ch[i] != '+' && ch[i] != '-' && ch[i] != '*' && ch[i] != '/' && ch[i] != ' ') {

```

```

// value에 값 할당
value = (ch[i] - '0');

if (ch[i + 1] >= '0' && ch[i+1] <= '9') {
    while (ch[i + 1] != NULL && ch[i + 1] >= '0' && ch[i] <= '9') {
        i++;
        // 자릿수 증가
        value *= 10;
        value += (ch[i] - '0');
    }
    if (ch[i + 1] != NULL && ch[i + 1] == '.') {
        // i를 증가시켜서 '.'으로 index 이동
        i++;
        // 소수점 처리를 위한 decimal 선언
        decimal = 1;
        // 소수점을 저장하기 위한 d_value 선언
        d_value = 0;
        // data가 NULL이 아니면서, 숫자 값인 경우 소수점의
아래자리로 처리

        while (ch[i + 1] != NULL && ch[i + 1] >= '0' && ch[i +
1] <= '9') {

            // index 증가
            i++;

            // d_value에 char값-'0'을 하여 char에서
숫자로 값 전환

            d_value = ch[i] - '0';

            // decimal의 갯수만큼 소수점 자리이동을
위한 for루프

```


d_value의 값은 $10^{(-1)}$ 배가 된다.

```
for (int j = 1; j <= decimal; j++) {  
    // 루프를 한번 순회할때마다  
  
    d_value *= 0.1;  
  
}  
  
// value에 소수점조정이 된 d_value를 누적합  
value += d_value;  
  
// decimal 증가  
decimal++;  
  
}  
  
// 연산을 위한 stack에 값 fpush  
fpush(&s, value);  
  
}  
  
else {  
  
    fpush(&s, value);  
  
}  
  
}  
  
else if (ch[i + 1] == '.') {  
  
    // i를 증가시켜서 '.'으로 index 이동  
  
    i++;  
  
    // 소수점 처리를 위한 decimal 선언  
  
    decimal = 1;  
  
    // 소수점을 저장하기 위한 d_value 선언  
  
    d_value = 0;  
  
    // data가 NULL이 아니면서, 숫자 값인 경우 소수점의  
아래자리로 처리  
  
    while (ch[i+1] != NULL && ch[i+1] >= '0' && ch[i+1] <= '9') {  
  
        // index 증가
```

전환

```
i++;  
  
// d_value에 char값-'0'을 하여 char에서 숫자로 값
```

for루프

```
d_value = ch[i] - '0';  
  
// decimal의 갯수만큼 소수점 자리이동을 위한
```

10⁽⁻¹⁾배가 된다.

```
for (int j = 1; j <= decimal; j++) {  
  
    // 루프를 한번 순회할때마다 d_value의 값은
```

```
        d_value *= 0.1;
```

```
    }  
  
    // value에 소수점조정이 된 d_value를 누적합
```

```
    value += d_value;
```

```
    // decimal 증가
```

```
    decimal++;
```

```
}
```

```
// 연산을 위한 stack에 값 fpush
```

```
fpush(&s, value);
```

```
}
```

```
else {
```

```
    // 소수점이 아닌 경우 바로 값 fpush
```

```
fpush(&s, value);
```

```
}
```

```
}
```

```
// 공백은 항을 구분하기 위함이므로, 연산을 진행하지 않음
```

```
else if (ch[i] == ' ') {
```

```

    }

    // 항목이 연산자

    else {

        // stack의 위에 위치한 값을 오른쪽 피연산자로 사용

        op2 = fpop(&s);

        // 오른쪽 피연산자의 아래 위치한 값을 왼쪽 피연산자로 사용

        op1 = fpop(&s);

        // 연산자 확인

        switch (ch[i])

        {

            // 연산자에 맞는 연산진행하여 stack에 fpush하여 값 저장

            case '+': fpush(&s, op1 + op2); break;

            case '-': fpush(&s, op1 - op2); break;

            case '*': fpush(&s, op1 * op2); break;

            case '/': fpush(&s, op1 / op2); break;

        }

    }

    // index 증가

    i++;

}

// fpop의 결과를 return

return fpop(&s);

}

```

/*

함수명 : main

인 자 : void

리턴형 : int

설 명 : main 함수, 실행시 최초로 실행되는 함수

*/

```
int main(void) {
```

```
    // 식을 입력받기 위해 char pointer ch 선언
```

```
    char *ch = (char *)malloc(sizeof(char)*100);
```

```
    // 사용자로부터 중위식을 입력받음
```

```
    printf("중위식을 입력하세요: ");
```

```
    scanf("%s", ch);
```

```
    // 1. 검사
```

```
    // 괄호 검사 후 return이 0이면 오류사항 출력 및 자원반환을 한다.
```

```
    if (bracketCheck(ch)) {
```

```
    }
```

```
    else {
```

```
        fprintf(stderr, "괄호에 오류있음\n");
```

```
        exit(1);
```

```
    }
```

```
    // 2. 변환
```

```
    // 중위식->전위식
```

```
    // 후위식이랑 반대개념
```

```
    printf("전위식: ");
```

```
    if (infix_to_prefix(ch) == -1) {
```

```
        printf("부호오류발생\n");
```

```
    }
```

```
    else {
```

```

        printf("%s\n", infix_to_prefix(ch));
    }

    // 중위식->후위식
    printf("후위식: ");
    if (infix_to_postfix(ch) == -1) {
        printf("부호오류발생\n");
    }
    else {
        printf("%s\n", infix_to_postfix(ch));
    }

    // 3. 계산
    // 후위식 계산
    printf("계산결과: ");
    if (infix_to_postfix(ch) == -1) {
        printf("부호오류발생\n");
    }
    else {
        printf("%lf\n", postfix_calculate(infix_to_postfix(ch)));
    }

    return 0;

}

```