자료구조 과제 2 – 배열 보고서

20222663 권기영

**1. 다항식 : 방식 1 – 모든 차수의 계수를 저장**

**<문제 내용>**

두 개 식의 계수를 차례대로 입력 받아서 두 개 식을 서로 더하고, 곱해서 그 결과가 되는 식을 새로 만들고, 식과 식에 값을 대입한 결과를 출력하는 문제이다.

입력의 개수가 정해져 있지 않아 scanf를 통해 정수로 입력을 받지 못하기 때문에, 문자열로 받은 후 어떻게 그 문자열들을 정수로 바꿔서 저장할지가 관건일 것이라고 생각한다.

**<해결 방안>**

1) 수식 1과 수식 2 입력 받아서 배열에 저장

fgets 함수를 활용하여 입력 한 줄을 전부 str이라는 배열에 저장한다. 그리고 str을 처음부터 끝까지 하나씩 배열요소를 호출하면서, 배열요소가 공백이거나 str이 끝까지 갔을 때 까지 새로운 배열 str\_i를 만들어서 배열요소를 저장한다. 그 후, 문자열로 저장된 입력 숫자 하나하나를 atoi(str\_i)를 통해서 수식의 계수(정수형)를 저장하는 배열인 p1, p2에 순서대로 집어넣는다.

입력되는 값은 6개가 아닐 수도 있지만, p1, p2에 저장하는 값은 6개가 되어야 한다. 예를 들어 최고차항의 차수가 3차인 식을 입력하는 경우 3차부터 입력하기 떄문에 입력되는 숫자는 4개지만, 배열에는 4차와 5차에 해당하는 위치에 0을 저장해서 총 6개를 저장해야 한다. 이를 구현하기 위하여 reverse함수를 만들어 사용하였다.

입력 : 3 0 6 3을 예로 들면, 일단 p1에 3 0 6 3을 차례대로 저장한다. 저장할 때 마다 count++을 해서 배열의 길이를 계산한다. 그 후 reverse 함수를 p1에서 count길이 만큼만 사용해서 p1의 요소를 3 6 0 3 으로 만든다. 이렇게 만들면 인덱스 값과 차수가 일치하게 된다. 그 후에, count 번째 요소부터 마지막 요소까지 0을 채워주면 최고차항의 차수를 넘는 인덱스값에는 0이 채워지게 된다.

reverse함수를 활용하지 않아도 상관없지만, reverse함수를 통해 인덱스 값과 차수를 일치시켜 좀 더 편하고 가독성 있게 코드를 작성하였다.

2) void add(int \*p1, int \*p2, int \*p3) - 더하기 구현(수식 1 + 수식 2)

인자로 p1, p2, p3를 받아서, p1과 p2의 0번째 요소부터 6번째 요소 까지를 더해서 p3의 요소에 차례대로 저장하는 add함수를 만들어 더하기를 구현하였다. **p3[i] = p1[i] + p2[i];**

3) void mul(int \*p1, int \*p2, int \*p4) – 곱하기 구현(수식1 \* 수식2)

인자로 p1, p2, p4를 받는다. 여기서 p4의 크기는 11인데, 그 이유는 5차식 \* 5차식을 하면 최대 10차식까지 나올 수 있기 때문에 배열의 크기가 최소 11이어야 전부 저장이 가능하다.

곱하기를 할 때 차수는 더해진다. 즉, 1차와 2차를 곱하면 차수는 (1 + 2) = 3이 된다. 이를 활용하여, p1과 p2 두 배열의 끝까지 반복되는 이중 for문을 활용하여, p1의 차수는 i, p2의 차수는 j일 때, p4의 차수는 i+j가 된다. 따라서 **p4[i+j] = p1[i] \* p2[j];**가 된다. 이는 인덱스 값과 차수의 값이 같기 때문에 가능하다.

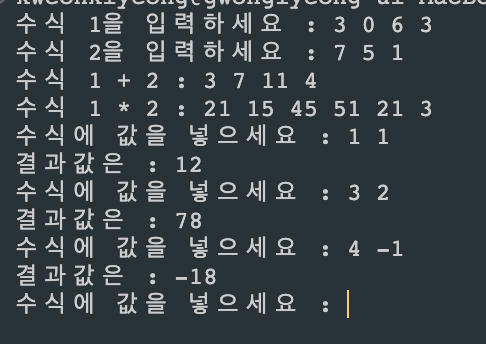
4) int eval(int \*arr, int len, int num) – 식에 값 대입한 결과 계산

인자로 배열 arr와 정수 len, num 총 3가지를 받는다. arr는 값을 대입할 식이다. num은 대입할 값이다. len은 그 식의 길이이다. 항 하나하나를 계산할 때, 계수 \* (대입한 값) ^ 차수 의 방식으로 계산하고, 모든 항의 값이 +로 이어져 있기 때문에, 각 항의 계산 결과를 모두 더해주면 된다, 즉, arr의 처음부터 끝까지 반복문을 돌려서 **result += arr[i] \* pow(num, i);** 로 결과를 계산할 수 있다.

5) 수식 3, 수식 4, 대입 결과 출력

식의 최고 차수만큼 반복되는 반복문을 활용하여 p3과 p4의 계수를 전부 차례대로 출력하면 수식 3과 수식 4의 결과를 출력할 수 있다. 대입 결과는, eval 함수의 인자 중에서 arr와 num에 해당하는 값을 scanf로 입력받아서 입력값에 해당하는 result를 출력하면 된다.

**<결과 출력>**



**2. 다항식 : 방식 2 – 계수가 0이 아닌 계수와 차수만 저장**

**<문제 내용>**

수식은 차수로 정렬된 상태로 입력 받고, 배열에 계수와 차수를 저장하여 더하기, 곱하기, 식에 값 대입의 연산을 구현하는 것이 목표이다. 방식 2로 하는 이유는 계수가 0인 항은 저장을 하지 않아서 공간을 아끼겠다는 목적이기 때문에, 저장할 항의 개수 \* 2만큼malloc으로 배열의 크기를 동적할당 하는 것이 중요하다. 또한 방식 1과 달리 계산하는 방식이 단순히 index만 맞춰서 계산할 수 있는 것이 아니기 때문에 계산을 어떻게 구현할 지가 주된 문제이다.

**<해결 방안>**

1) 수식 1과 수식 2 입력 받아서 저장

계수(coef)와 차수(degree) 총 2가지 종류를 저장해야 하기 때문에 이 둘을 멤버로 가지는 struct Poly를 선언하여 저장했다.

방식 1에서 구현했던 방식과 달리, 방식 2를 구현할 때는 strtok의 존재를 알게 되어서, 공백을 토큰으로 문자열을 하나씩 구분해서 tok 에 저장하고, tok을 atoi를 통해서 정수로 변환하는 방식으로 입력을 받았다. 이 때, 계수가 0이면 계수와 차수를 저장하지 않아야 한다.

fgets로 한 줄을 저장하고, 반복문을 통해 처음부터 끝까지 반복하면서, 수식을 저장하는 상황마다 count++하였고, 입력에서 계수는 무조건 0번째, 2번째 … 즉 짝수 번째에 나오므로 (count % 2 == 0)이면 계수가 입력 받는 것, (count % 2 == 1)이면 차수가 입력 받는 것이다. 따라서 입력 값이 0일 때, count % 2 == 0 이라면 계수가 0이므로 현재 입력과 다음 입력을 저장하지 않고 넘기는 것이다.

또한 위에서 입력 받은 값들을 구조체에 바로 저장하지 않고 일단 temp라는 임시 문자열에 count / 2로 개수를 맞춰 동적 할당할 수 있고, 동적할당을 한 후에 temp의 배열요소들을 다시 구조체 Poly로 옮겨주면 된다. **Poly[i/2].coef = temp[i]; Poly[i/2].degree = temp[i+1];**

2) void add(int \*p1, int \*p2, int \*p3) – 더하기 구현

방식 1과 달리, 방식 2의 구현은 조금 복잡하다.

p3에 더하기 연산 결과를 저장하기 전에, p3의 크기를 먼저 동적할당 해주어야 하는데, check\_degree라는 새로운 배열을 만들어서 p1과 p2에 무슨 차수 종류가 있는지 저장한다. 값이 0이면 없는 차수 인거고, 값이 0이 아니면 있는 차수이므로, 있는 차수 종류 개수를 count해서 count의 크기만큼 p3를 동적할당 해준다.

수식 1의 인덱스를 i, 수식 2의 인덱스를 j로 하고, 둘 중 하나의 수식이 끝까지 갈 때 까지 while문을 돌린다. 서로의 차수를 비교하여, 수식 1의 차수가 높으면 p3에 p1[i].coef, p1[i].degree를 저장하고 i++, 수식 2의 차수가 높으면 p3에 p2[j].coef, p2[j].degree를 저장하고 j++, 차수가 서로 같으면 p3에 p1[i].coef + p2[j].coef를 계수로 저장하고, 차수는 같으니까 그냥 아무거나 저장, i++, j++ 한다. 둘 중 하나의 수식이 끝나면 다른 하나의 수식에 남은 계수와 차수를 전부 p3에 차례대로 저장해준다.

3) void mul(int \*p1, int \*p2, int \*p4) – 곱하기 구현

p4에 곱하기 연산 결과를 저장하기 전에, p4의 크기를 먼저 동적할당 해주어야 한다. 위의 더하기 구현에서 p3를 동적할당 했던 방식과 같이, check\_degree 배열을 활용하여 차수의 종류를 센다. 이 때, 곱하기에서 차수는 더해지므로 p1[i].degree + p2[j].degree의 종류를 count 해준다.

곱하기의 구현에서, 중요한 것은 처음 나오는 차수인지, 중복되는 차수인지가 중요하다. 즉, 처음 나오는 차수의 계산은 그냥 계산된 값을 저장하면 되지만, 두 번째 부터는 이미 저장된 결과에 더해줘야 한다.

p4에 새로운 차수를 저장할 때 마다 cnt++을 해주고, 연산이 시행될 때 마다 cnt번 반복을 통해서 p4에 저장된 모든 차수들과 현재 시행하는 연산의 차수를 비교해서 새로운 차수이면 check.= 0, 이미 나온 차수이면 check = 1로 바꾸는 식으로 구현한다. 이를 통해서 check = 0이면 계수와 차수 값을 그냥 저장, check = 1이면 저장된 계수에다가 더해주는 식으로 구현하였다.

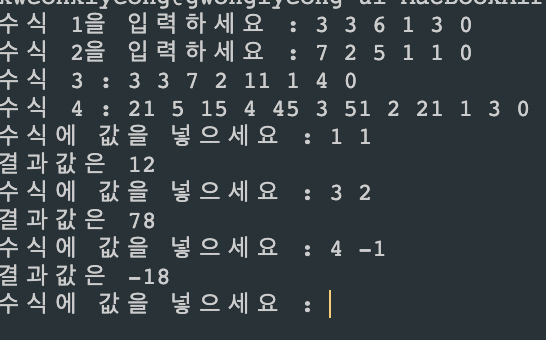
4) 식에 값 대입한 결과 계산

방식 1과 거의 똑같은 방식으로 항 하나의 값 = 계수 \* 대입한 값 ^ 차수 의 형식으로 계산한 뒤 각 항을 모두 더해주는 방식이다. 따라서 **result += p[i].coef \* pow(num, p[i].degree);**

5) 수식 및 결과 대입 결과 출력

수식은 그냥 저장된 계수와 차수를 차례대로 출력했고, 대입 결과도 입력된 수식과 대입 값에 따라 그에 맞는 결과값을 출력하도록 구현했다.

**<결과 출력>**



**3. 파이썬의 리스트 이해하기**

1) 파이썬의 리스트를 어떻게 c언어로 구현했을까?

파이썬의 리스트는 type의 정보를 가지는 멤버와 void 포인터 멤버 2가지를 가지는 구조체배열로 구현이 되었을 것 같다. 예를 들어, int type; (void \*)data; 두 가지의 멤버를 가지는 구조체배열이 있다. 그래서 if (type == 0) -> char 형, type == 1 -> int 형. 이런 식으로 type 멤버로 자료형을 처리한다. 또한 (void \*)data는 자료의 첫 번째 주소를 나타낸다. 그러나 주소만 알면 몇 바이트를 읽어야 하는지 모르므로, type 에서 자료형 정보를 받아 자료형에 맞는 바이트를 읽어서 저장한다.

예를 들어, 파이썬 리스트에서 0 번째 배열의 데이터 값은 \*((char \*)struct[0].data) 이고, 데이터가 char 형이라면 struct[0].type = 0; 이 저장되어 1byte를 읽어야 한다는 것을 알 수 있을 것이다.

2) 파이썬 리스트 메소드 요약 및 구현 방법 예측

(1) append – 리스트에 요소 추가

요소를 추가할 때 마다 동적으로 저장 공간을 추가 할당 해주고, for문으로 끝까지 count 하여 list의 길이를 구한 뒤, 마지막 index + 1의 위치의 배열요소에 추가할 요소를 저장한다.

(2) sort – 리스트 정렬

정렬 알고리즘에 대한 사전 지식이 없었다면 기껏해야 버블 정렬이나 삽입 정렬 정도를 생각했을 것이다. 그러나 퀵 정렬이 제일 빠른 정렬 알고리즘으로 알고 있기 때문에, 분할 정복 기법을 활용한 퀵 정렬으로 구현했을 것으로 생각한다.

(3) reverse – 리스트 뒤집기

reverse 함수는 c언어를 할 때도 많이 만드는 함수이다. for 문으로 끝까지 count 해서 리스트의 길이를 구하고, 길이 / 2 만큼 for문을 돌려서 list[i] 와 list[len – i – 1]의 데이터를 swap해주면 된다.

(4) index – 인덱스 반환

어떤 data가 있으면,while (list[i] != data) i++; 즉, 그 data가 나올 때 까지 index를 증가시키다가 data가 나오면 index를 return한다.

(5) insert – 요소 삽입

C언어에서 삽입을 하기 위해서는 삽입하려는 index 오른쪽에 있는 모든 배열요소를 한 칸씩 오른쪽으로 옮겨야 한다. 이 때 주의할 점은, 배열의 오른쪽 끝 부터 옮겨야 한다. 왼쪽부터 오른쪽으로 옮기면 왼쪽에 있는 배열요소 값이 오른쪽에 있는 배열요소 값을 다 집어삼킨다.

(6) del – 리스트 요소 삭제

del 함수는 index 값을 지정하여 요소를 삭제한다. C언어에서 삭제를 하기 위해서는 삭제하려는 index 오른쪽에 있는 모든 배열요소를 한 칸씩 왼쪽으로 옮겨야 한다. Index + 1번째에 있는 요소부터 옮겨야 한다.

(7) remove – 리스트 요소 삭제

remove(x) 는 첫번째로 나온 x를 삭제하는 함수이다. 이는 while (list[i] != x) i++;을 하고, x가 나오면 그 인덱스 값을 기준으로 del을 구현하면 된다.

(8) pop – 마지막 요소 끄집어내기

배열의 길이를 구하고, 마지막 요소를 return 후 삭제하면 된다.

(9) count – 리스트 요소 개수 세기

For 문을 활용하여 리스트 처음부터 끝까지 count++ 한다.

**4. C에서 행렬 구현하기**

**<문제 내용>**

행렬을 자유롭게 사용할 수 있도록 행렬의 저장방식, 기능, 연산 등을 정의하고 구현한다. 행렬은 자유롭게 생성할 수 있고, 행렬의 저장은 효율적이어야 한다. 즉, 동적할당을 이용해 필요한 만큼만 공간을 할당해야 한다. 또한 행렬의 덧셈, 곱셈, 전치 등의 연산을 구현해야 한다. 행렬에 들어가는 값은 정수로 제한한다.

이 문제에서 가장 중요한 점은 바로 함수를 이용하고, 그 함수의 인자로 행렬을 받아야 한다는 점이다. 행렬을 동적할당 하기 위해서는 결국 이중 포인터로 선언하고, 매개변수로 이중 포인터를 넘겨줘야 한다는 것인데, 여기서 어려움이 생길 것이라고 예상했다.

또한 가장 문제였던 점은 바로 함수 안에서의 행과 열 크기 계산이었다. 행렬 포인터 만을 매개변수로 받고, 행과 열의 크기 값은 매개변수로 받지 않아도 함수 내에서 행렬 저장 정보만을 토대로 행과 열의 크기를 계산할 수 있는지가 문제였다. 이중 포인터가 아니라 2차원 배열로 선언된 행렬이었다면 sizeof를 활용할 수 있지만, 포인터는 포인터 메모리 공간으로 고정된 값만을 반환하기 때문에 sizeof를 사용할 수 없다.

**<해결 방안>**

1) 함수 내에서 행과 열 크기를 계산할 수 있도록 행렬을 동적할당 하기

행과 열을 각각 크기보다 한 칸씩 더 할당해서, 각 행의 마지막 열에는 ‘x’ 를 저장하고, 마지막 행에는 NULL포인터를 저장하는 것이다.

즉. row와 column 에 대해서, matrix[i][column] = ‘x’; matrix[row] = NULL; 이런 식으로 저장한다. 그런 뒤에 함수 내에서 column값을 계산할 때는 matrix[row][column] != ‘x’일 때 까지 column++ 해 주고, row값을 계산할 때는 matrix[row] != NULL일 때 까지 row++하는 방법으로 매개변수로 행과 열 크기 값을 전달하지 않고도 행렬만으로 행과 열의 크기를 계산할 수 있다.

물론 저장공간 측면에서 조금 더 비효율적이지만, 프로그램을 사용하는 입장에서 행렬을 다룰 때 마다 매번 행과 열 값을 입력해야 하는 것 보다 이게 더 편하다고 판단하여 이런 방식을 사용하였다.

2) Matrix createMatrix(int row, int column) – 행렬 생성

Matrix 자료형은 int\*\*이다.

row와 column값을 받아서 각각 행 = row + 1, 열 = column + 1만큼 동적할당 해서 2차원 배열을 만든 후, row \* column 크기에 0을 대입해준다. 함수의 자료형이 Matrix이므로 Matrix 자료형 행렬을 return 한다.

따라서 함수의 사용은 Matrix a; a = createMatrix(3,3); 과 같은 형식으로 사용한다.

3) void randMatrix(Matrix \*matrix, int start, int end) – 행렬에 난수 집어넣기

함수의 자료형은 void이고, 인자로 난수의 범위를 나타내는 변수 start, end를 전달받고, 난수를 집어넣고자 하는 행렬의 주소값을 전달받는다.

자료형이 Matrix인 함수들은 행렬을 return하지만, void인 함수들은 행렬을 return 하는게 아니라 이미 만들어져 있는 행렬의 상태를 바꾸는 것이기 때문에 행렬의 주소값을 인자로 받아야 한다.

프로그램을 실행할 때 마다 다른 난수를 생성하기 위하여 srand(time(NULL)); 을 사용하였고, 각 행렬요소에 **rand() % (end – start + 1) + start** 를 집어넣었다. 예를 들어, start = 2, end = 10이라고 하면, rand() % 9 + 2 이므로 2부터 10 까지의 난수를 집어넣을 수 있다.

4) Matrix createMatrixfromArray(int \*arr, int row, int column) – 1차원 배열의 데이터를 바탕으로 새로운 행렬 생성

새로운 행렬을 선언한 뒤, 입력받은 1차원 array로 부터 값을 꺼내 행렬에다가 집어넣은 뒤 그 행렬을 return 하는 함수이다.

일단 입력받은 row, column의 값을 토대로 새로운 행렬 temp를 함수 내에서 선언한다. 그 다음에는 temp[i][j] = arr[i \* column + j];의 코드를 통해서 arr의 값들을 차례대로 행렬 temp에 저장한다. 그리고 temp를 return 한다.

이 함수는 1차원 배열 arr의 크기가 row \* column 이어야 의도대로 동작하도록 만들어졌다.

5) Matrix resizeMatrix(Matrix \*matrix, int row, int column) – 행렬의 규격 바꾸기

규격이 다른 행렬을 선언하고, 그 행렬에 원래 있던 데이터를 차례대로 집어넣고, 데이터가 부족하면 남는 부분에 0을 저장하고, 데이터가 넘치면 있는 데이터를 버린다. 그 후에 행렬을 return한다.

입력으로 받은 행렬이 matrix이고, return 하려고 하는 새로운 행렬이 temp 이다. 우선 입력 받은 row와 column 값에 맞게 temp를 동적할당 해준다. 그 후에 temp의 모든 원소에 0을 집어넣는데, 그 이유는 어짜피 데이터가 부족하면 남는 부분에 0을 저장해야 하므로 미리 저장해놓는 것이다.

그리고 matrix의 원소들을 1차원 배열 arr에 넣는 작업을 수행한다. 두 행렬 matrix와 temp는 규격이 다르므로 데이터를 주고받으려면 인자 설정이 매우 불편하겠다고 생각하였고, 그래서 matrix의 데이터를 1차원 배열 arr에다가 집어넣고, matrix[i][j] = arr[count++]; 를 해서 arr의 데이터를 matrix에 집어넣도록 구현하였다.

인자 matrix를 주소값의 형태로 받은 이유는, 원래 있던 행렬 matrix의 동적할당을 함수 내에서 free해주기 위함이다. 따라서 free(\*matrix)로 원래 행렬을 없애고, 새로운 행렬 temp를 return 한다. 사실상 원래 행렬을 resize하는 것이 아니라, 행렬을 없앤 뒤에 다른 규격의 데이터만 같은 행렬을 새로 만드는 것이다.

6) Matrix addMatrix(Matrix a, Matrix b) – 두 행렬을 더한 새로운 행렬 생성

행렬합을 수행하기 전에, 두 행렬의 규격이 같은지를 먼저 판단하고, 두 행렬의 규격이 다르다면 return NULL; 하는 방식으로 예외를 처리해 주었다.

두 행렬과 규격이 같은 새로운 행렬 temp를 동적할당으로 선언하고, **temp[i][j] = a[i][j] + b[i][j];** 로 같은 위치의 원소를 서로 더해서 temp에 저장하였다. 그 후에 temp를 return한다.

7) Matrix multiplyMatrix(Matrix a, Matrix b) – 두 행렬을 곱한 새로운 행렬 생성

행렬곱을 수행하기 전에, column\_a = row\_b 일 때만 행렬 곱을 수행할 수 있으므로 이를 만족하지 않는다면 return NULL; 하는 방식으로 예외를 처리해 주었다.

A 행렬의 규격을 (i \* k), b 행렬의 규격을 (k \* j)라고 하면, i, j, k를 인자로 하는 3중 for문을 활용하여 행렬곱을 구현할 수 있다. temp의 규격은 (i \* j) 이므로 **temp[i][j] += a[i][k] \* b[k][j];**의 형식으로 a행렬의 i 행의 원소들과 b 행렬의 j 열의 원소들을 전부 더해서 temp의 (i, j)위치의 원소에다가 저장한다.

8) Matrix transposeMatrix(Matrix \*matrix) – 전치 행렬 생성

처음에는 전치 연산을 수행하기 위해서 인자만 바꿔서 넣으면 될 것이라고 생각하였으나, 반복을 진행하는 도중에 실시간으로 원소가 바뀌어서 연산이 제대로 수행되지 않는 문제가 발생하였다.

이를 해결하기 위해서 인자로 전달받은 matrix 와 똑같은 규격의 새로운 행렬 matrix\_t를 새로 생성하여 matrix\_t에다가 matrix의 전치를 저장하면 연산을 수행하는 도중에 행렬이 바뀌어버리는 문제가 해결된다. **matrix\_t[i][j] = matrix[j][i];** 와 같은 방식으로 전치 연산을 수행하고, matrix\_t를 return한다.

9) void printMatrix(Matrix matrix) – 행렬 출력

행렬을 print하는 함수는 매우 쉽게 구현할 수 있다. 일단 함수 안에서 matrix의 row값과 column 값을 구한다. 2중 for문을 사용하여, 열은 공백으로 구분하고, 한 행이 끝나면 printf(“\n”); 을 해서 행을 구분하는 식으로 출력한다.

**<결과 출력>**

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

**5. 계산기**

**<문제 내용>**

중위 표기법으로 수식을 입력하면 그 수식을 후위 표기법으로 변환한 뒤에 결과를 계산하여 출력해라. 또한 잘못된 수식을 입력하면 오류를 찾아서 표기해라.

사용되는 연산자는 +, -, \*, /, \*\*(지수연산자), ( ) 이다. 또한 지수 연산자가 연속해서 두 번 나오면 우결합으로(오른쪽에서 부터) 계산한다. 입력으로 사용되는 값은 실수 자료형이다. 이러한 조건을 가진 계산기를 stack을 활용하여 구현한다.

일단 첫 번째 문제는 자료형이 실수형인 것이다. C언어를 사용하기 때문에 문자열 한 줄을 전부 받은 뒤, 이를 어떻게 parsing할 지에 대한 방법이 중요할 것 같다. 두 번째 문제는 오류 출력이다. 사용자의 입력은 매우 다양할 수 있으므로 입력에 대한 오류는 매우 많을 것으로 예상되는데, 그러한 오류들을 어떻게 예외 처리하고 오류를 표기할지에 대한 고민이 필요하다

**<해결 방안>**

.1) stack의 구현 – push, pop, peek, isEmpty, isFull 함수

Stack은 char 형 배열 stackarr[]와 top을 멤버로 가지는 구조체로 선언하였다. 또한 top의 초기 상태는 -1로, 가장 최근에 쌓인 값을 가리키는 인덱스 값으로 정의하였다.

**isEmpty** : top == -1이면 True, 아니면 False 반환.

**isFull** : top == MAX\_SIZE – 1 이면 True, 아니면 False 반환.

**push** : isFull === True 이면 오류 메세지 출력. 아니면 top++; 하고 값 저장.

**pop** : isEmpty = True 이면 오류 메세지 출력. 아니면 값을 return 하고 top--;

**peek** : isEmpty == True 이면 오류 메세지 출력, 아니면 값을 return.

2) isOperator, Priority 함수 – 연산자 판별, 우선순위 반환 함수

**isOperator** : 문자를 인자로 받아서, 그 문자가 연산자이면 return True, 아니면 return False.

**Priority** : 문자를 인자로 받아서, 문자가 +, - 이면 return 1; 문자가 \*, / 이면 return 2; 문자가 ^ 이면 return 3; default 이면 return 0;

3) void InfixtoPostfix(char \*infix, char \*postfix) – 중위표기식을 후위표기식으로 변환하는 함수.

infix는 사용자가 중위표기법으로 입력한 수식이 들어있다. Infix 문자열의 문자 각각을 처음부터 끝까지 확인하면서, 연산자일 때와 피 연산자일 때로 구분하여 처리한다. 후위표기식은 postfix에다가 저장한다.

**(1) 문자가 피 연산자일 때**

temp에다가 문자를 받는다. 만약에 isdigit(temp) == True 이거나, temp == ‘.’ 이면 문자가 피 연산자 이다. 실수형이기 때문에 ‘.’도 피 연산자로 처리해야 한다. temp에 피 연산자가 들어오게 되면, infix[i] != ‘ ‘(공백이 되기 전 까지)일 동안 postfix[index++] = infix[i++]; 해서 피 연산자를 postfix에 바로 넘겨준다.

숫자를 입력 받으면 바로 후위표기식에 적으면 되므로 바로 postfix로 간다.

**(2) 문자가 연산자일 때**

연산자를 저장하기 위해서 stack을 생성한 것이다.

isOperator(temp) == True 라면, 문자가 연산자일 때 이다. 이 때는 (인 경우, )인 경우, (+, -, /)인 경우, \*인 경우 총 4가지로 나누어서 생각한다.

먼저 (인 경우에는 무조건 stack에 push한다.

)인 경우에는 stack의 top에 (가 나올 때 까지 pop 해서 postfix에 적는다.

+, -, / 인 경우에는 Priority 함수를 활용해 stack의 top에 있는 연산자와 비교해서, temp의 우선순위가 stack의 top에 있는 연산자의 우선순위 이하라면 postfix에 pop한다. 모두 pop한 뒤에 temp를 push한다.

\*인 경우에는 두 가지 case로 나뉜다. 만약에 infix[i+1] = ‘\*’ 인 경우에는 입력으로 \*\*를 입력한 것이므로 stack에 ^를 push 한다. 그렇지 않다면 위의 +, -, /의 경우와 같이 처리한다.

Infix를 모두 돌았는데 isEmpty(&stack) == False라면 stack에 남은 연산자들을 차례대로 postfix에 pop 해 준다.

모든 연산자와 피 연산자는 공백을 기준으로 구분되도록 postfix에 push하였다.

4) double Evaluate(char \*postfix) – 후위표기식을 받아서 계산하는 함수

postfix 에는 후위표기식으로 작성된 수식이 공백을 기준으로 구분되어 들어있다. postfix에 저장된 문자를 하나하나 끝까지 확인하면서, 연산자인지 피 연산자인지에 따라 다르게 처리한다.

또한 stack은 피 연산자를 저장하기 위한 목적으로 생성한다.

**(1) postfix[i]가 피 연산자일 때**

Isdigit(postfix[i]) == True 이거나 postfix[i] = ‘.’일 때, 피 연산자이다. 이 때는 피 연산자일 동안 계속 stack에 push한다. 마지막에 공백을 push함으로써 서로 다른 숫자를 구분한다.

**(2) postfix[i]가 연산자일 때**

isOperator(postfix[i]) == True이면 연산자이다. 이 때는 stack에 들어간 피 연산자에 대한 처리가 필요하다. 문자열로 stack에 저장되어 있는 피 연산자를 실수형으로 저장할 변수 operand1, operand2를 선언한다. stack에 저장되어 있는 문자를 공백이 나올 때 까지 하나씩 받으면서, temp\_f라는 문자열에 저장한다. stack의 위에서부터 받았으므로 피 연산자는 거꾸로 정렬 되어있는 상태이기 때문에 reverse함수를 만들어서 다시 제대로 정렬해 준다. 그 후 operand 함수에 atof(temp\_f)를 저장한다. atof는 문자열을 실수형으로 바꿔주는 함수이다.

stack에 문자열로 저장되어 있던 피 연산자들을 실수형으로 바꿔 operand1, operand2에 저장했으면, 이제 연산자의 종류에 따라 실행이 달라진다.

연산자의 종류에 맞춰서 answer에 계산하여 저장한다. 여기서 예외적으로 ^가 연산자인 경우에는, ^가 2개인지 1개인지에 따라서, 처리가 달라진다. ^가 1개만 나왔다면 그냥 pow(operand2, operand1) 연산을 수행한다. ^가 2개 연속으로 나왔다면operand3이라는 변수를 하나 더 만들어서 stack에 있는 피 연산자를 하나 더 꺼내서 저장한다. 그 후에 pow(operand3, pow(operand2, operand1)); 연산, 즉 우결합 지수 연산을 수행하여 answer에 저장한다.

연산이 한번 끝났다면 answer을 sprint 함수를 활용하여 다시 문자열로 바꾸고, stack에 push한다.

**(3) 모든 계산이 끝난 뒤 stack에 남은 숫자 처리**

모든 계산이 끝난 뒤 stack에 남은 문자열은 결과값일 것이므로, 이를 atof를 통해 형변환을 거친 뒤 return 한다.

5) 결과값을 소수점 몇 자리까지 제시하는 것이 좋은가?

그냥 %f으로 실수를 출력하면 소수점 아래 6자리 까지 출력된다. 필요 없는 소수점들을 제거할 방법에 대해서 고민하던 와중, %g의 존재에 대해서 알게 되었다. %g로 실수를 출력하게 되면 필요 없는 소수점들을 모두 제거하고 출력 할 수 있다!

6) void Check\_Error(char \*infix) – 사용자의 입력에서 오류를 확인하고 출력하는 함수

오류가 검출되면, infix[i]로 처음부터 끝까지 확인하기 때문에 errorplace = i; 로 오류가 생긴 위치를 받고, 그 위치에 오류를 출력한다.

**(1) 괄호의 개수가 맞지 않을 때**

Infix를 처음부터 끝까지 확인하면서, 여는 괄호가 나오면 count++, 닫는 괄호가 나오면 count-- 해서 괄호 쌍의 개수를 세준다. 여는 괄호가 무조건 닫는 괄호보다 먼저 나오므로 만약에 count가 음수가 되면 )가 하나 더 많은 것이므로 바로 오류를 출력한다.

Infix를 끝까지 확인했을 때, count == 0 이면 오류가 없는 것이고, count가 양수라면 (가 더 많은 것이므로 오류를 출력한다. 오류의 위치 errorplace = strlen(infix) – count; 이다.

**(2) 연산자가 두 번 연속으로 들어올 때**

이 오류는 예외사항이 3 가지 있다. 첫 번째 : 연산자 다음에 (가 올 때, 두 번째 : ) 다음에 연산자가 올 때, 세 번째 : \*\*가 입력으로 들어올 때. 이 3가지의 예외처리를 해야 오류를 정상적으로 출력할 수 있다.

isOperator(infix[i]) == True 이면, check\_oper = 1; 을 저장한다. Isdigit(infix[i]) == True 이거나 infix[i] == ‘)’ 라면 그 다음에 연산자가 나와도 2번 연속이 아니기 때문에 check\_oper = 0; 을 저장한다. 만약 연산자가 나왔을 때, check\_oper == 1; 이라면 연산자가 두 번 연속으로 들어온 것이기 때문에 위의 3가지 예외를 제외하고는 오류를 출력한다.

**(3) 숫자 사이에 공백이 들어올 때**

check\_digit을 생성해서 피 연산자가 나오면 check\_digit = 1; 로 저장하고, 연산자가 나오면 check\_digit = 0;로 저장한다. 그 후에 isdigit(infix[i]) == True; 즉 피 연산자가 나왔을 때, check\_digit == 1; 이면 오류를 출력한다.

**(4) \*연산자 사이에 공백이 있을 때**

\*\*가 입력으로 주어진다면 지수연산자 ^로 받지만, \* \*가 입력으로 주어지면 \*연산자가 두 개 연속으로 나온 것이므로 오류를 출력해야 한다. 위에서 연산자가 두 개 연속으로 나왔을 때 오류를 출력하는 코드에서, \*가 두 개 연속으로 나오는 경우는 예외처리 하였으므로 여기서 다시 오류를 검출해준다.

이것 또한 위의 코드와 비슷하게 check\_mul을 생성해서, \*가 나오면 check\_mul = 1; 피연산자가 나오면 check\_mul = 0; 으로 저장한다. \*가 다시 나왔을 때 check\_mul == 1 이고 infix[i-1] == ‘ ‘ 이라면, 즉 \* 사이에 공백이 있다면 오류를 출력한다.

이것 외에도 여러가지 예외사항(연산자, 피연산자로 정의되지 않은 ?, # 등이 입력되었을 때 등)들이 있지만, 나올 수 있는 오류는 너무 많고, 그 모든 오류를 검출하는 것은 불가능하다고 판단되어 과제 내용에 있거나, 수업시간에 언급한 오류 위주로 구현하였다.

**<결과 출력>**

**텍스트이(가) 표시된 사진

자동 생성된 설명**