자료구조(Data Structure)

Programming Assignment 3

20161603
신민준

목차

1. 문제1

문제 1은 infix에서 postfix로 식을 바꿀 때, minus의 unary 연산자까지 포함해서 식을 postfix로 바꾸기를 요구합니다. 어떠한 연산자가 unary이기 위해서는 다음과 같은 조건 중 하나를 만족해야 합니다.

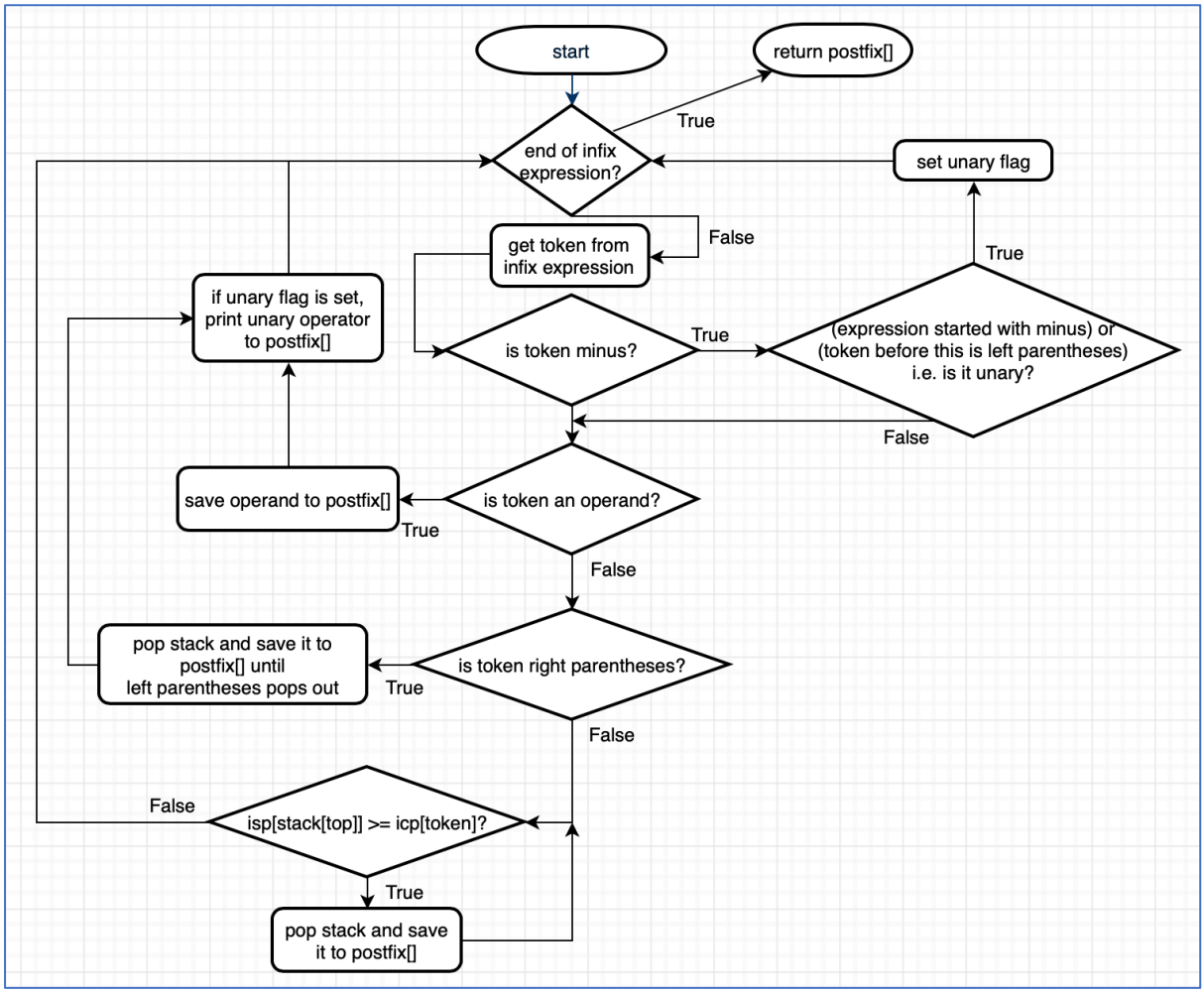
- 해당 연산자가 infix 식의 첫번째 element이거나
- Right parentheses(')') 문자를 제외한 연산자 바로 뒤에 해당 연산자가 옴.

따라서, 기본적인 infix to postfix translation 알고리즘에 이 조건을 확인하는 구문을 추가하는 방향으로 프로그램을 구성했습니다.

Postfix로 변환하는 프로세스는 to_postfix() 함수가 담당하며, 변환된 postfix 식을 계산하는 프로세스는 eval_postfix() 함수가 담당하도록 설계했습니다. 이 두 함수의 Flowchart는 다음과 같습니다.

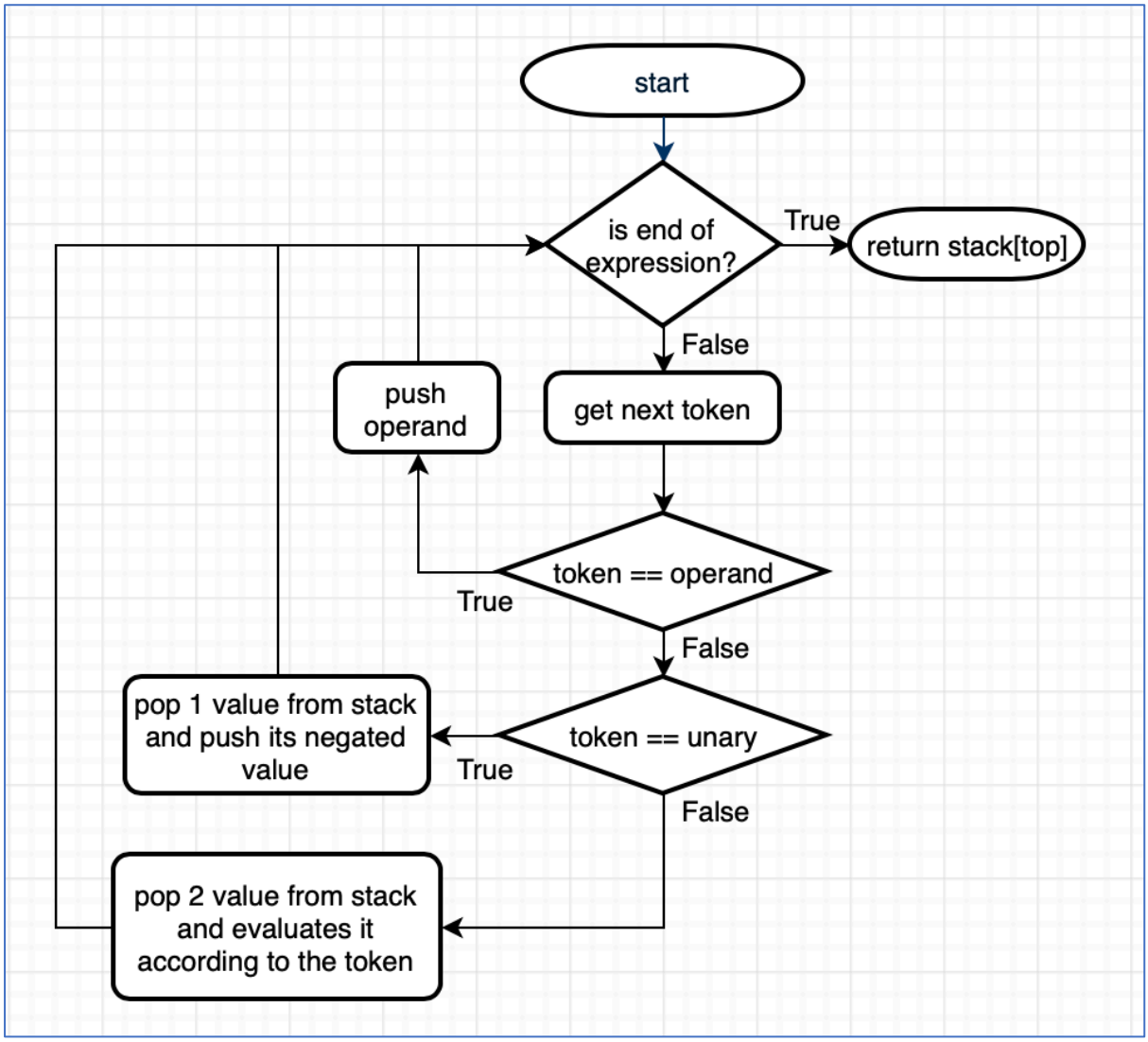
1.1 프로그램 구조

<to_postfix() function>



Flowchart 1

<eval_postfix() function>



Flowchart 2

1.2 세부 설명

함수 to_postfix()는 인자로 전달받은 infix expression을 한 글자씩 검토하면서 프로세스를 수행합니다. 각 loop 마다 해당하는 글자와 precedence의 index를 나타내는 token 값을 구합니다. token 값은 getToken() 함수를 사용해 구합니다.

```
/**
 * Does the opposite of getOp. Gets precedence value from given operation char.
 * @param  op Character that needs analysising.
 * @return    Corresponding precedence value for op.
 */
precedence getToken(char op) {
   switch(op) {
      case '(': return leftp; break;
      case ')': return rightp; break;
      case '+': return plus; break;
      case '-': return minus; break;
```

```
      case '*': return times; break;
      case '/': return divide; break;
      case '%': return mod; break;
      case '#': return unary; break;
      default: return operand; break;
   }
}
```

Code 1

먼저, 해당 token이 minus라면, unary일 가능성이 있기 때문에, 먼저 해당 minus가 unary인지 확인하는 작업이 필요합니다.

```
      if (token == minus) { // minus unary check
         if(i == 0) {
            f_unary = 1;
            continue;
         } else if(getToken(infix[i-1]) != operand &&
               getToken(infix[i-1]) != rightp) {
            f_unary = 1;
            continue;
         }
      }
```

Code 2

만일 minus가 1. Expression의 첫 원소이거나 2. ')' 연산자가 아닌 다른 연산자 바로 뒤에 위치해있다면, 해당 minus는 unary이므로, unary flag를 뜻하는 f_unary를 set하고, 바로 continue문을 사용해 다음 loop로 넘어갑니다.

해당 token이 minus가 아니라면, unary 검사는 필요가 없으므로 다음 단계로 넘어갑니다. token이 operand라면, 해당 operand를 결과 expression인 postfix[] 배열에 저장합니다. 이 때, 만약 unary flag가 설정되어있다면, unary 문자도 postfix[] 배열에 저장합니다.

```
      if(token == operand) {
         postfix[post_idx++] = op;
         if(f_unary) { //unary print process
            postfix[post_idx++] = getOp(unary);
            f_unary = 0;
         }
      }
```

Code 3

여기서 사용된 getOp() 함수는 token 값을 받아 해당하는 char 문자를 리턴하는 함수입니다.

```
/**
 * Get corresponding character for given precedence token
 * @param  token Precedence value that you wish to get character for
 * @return       Character for given precedence.
 */
char getOp(precedence token) {
   switch(token) {
      case leftp: return '('; break;
      case rightp: return ')'; break;
```

```
      case plus: return '+'; break;
      case minus: return '-'; break;
      case times: return '*'; break;
      case divide: return '/'; break;
      case mod: return '%'; break;
      case unary: return '#'; break;
      default: return -1;
   }
}
```

Code 4

token이 right parentheses라면, 지금까지 스택에 push한 operator 중 가장 최근의 left parentheses까지 전부 다 postfix[]에 저장합니다. 그 후, unary minus를 추가해야하는지 확인합니다.

```
      else if(token == rightp) { // token is right parentheses
         while(stack[top] != leftp) { //check for top validity
            // pop until left parentheses
            postfix[post_idx++] = getOp(pop(stack, &top));
         }
         pop(stack, &top); // pop left parentheses from stack
         if(f_unary) { //unary print process
            postfix[post_idx++] = getOp(unary);
            f_unary = 0;
         }
      }
```

Code 5

이 모든 case에 token이 들어가지 않았다면, token이 right parentheses 가 아닌 어떤 operator라는 뜻이므로, token의 precedence와 stack에서 최상단에 있는 element의 precedence를 비교해 token의 precedence가 stack[top]의 precedence보다 낮아질 때 까지 스택 안의 operator를 pop()하고 postfix[]에 저장합니다. 그 후, 스택에 현재 token을 push합니다.

```
      else {
         while(top != -1 && isp[stack[top]] >= icp[token]) {
            // while in-stack-precedence of stack's top
            //       >= incoming precedence of token
            postfix[post_idx++] = getOp(pop(stack, &top)); // pop stack
         }
         push(stack, &top, token);
      }
```

Code 6

지금까지의 process를 infix expression의 각 요소에 대해 모두 수행하면, 스택에 남아있는 모든 operator들을 postfix[] 배열에 저장하고, 이 배열을 리턴합니다.

```
while(top != -1) {
      // pop remaining operators
      postfix[post_idx++] = getOp(pop(stack, &top));
   }
```

```
postfix[post_idx] = '\0'; // for null-terminating string
return postfix;
```

Code 7

함수 to_postfix()의 시간복잡도는 infix expression의 길이를 n이라 할 때, $O(n)$입니다.
to_postfix() 함수를 계산하는 eval_postfix() 함수 또한, 기존의 postfix 계산 알고리즘에 unary operator에 대한 exception 처리만 추가한 형식입니다. Unary operator를 처리할 때는, 다른 operator와는 다르게, 인자를 하나만 받아야 하기 때문에, 따로 case 처리를 했습니다.

```
for(int i=0 ; i<(int)strlen(expression) ; i++) {
   op = expression[i];
   token = getToken(op);
   if(token == operand) {
      // push all operands into stack
      push_int(stack, &top, op-'0');
   } else if(token == unary) {
      // pop 1 operand if token is unary. evaluated value is pushed
      arg2 = pop_int(stack, &top);
      push_int(stack, &top, -arg2);
   }
   ...
}
```

Code 8

[Code 8] 은 unary operator가 들어왔을 때의 처리 과정을 보여주고 있습니다. 이 과정은 다른 operator가 처리되기 이전에 수행됩니다. 다른 operator의 처리과정은 다음과 같습니다.

```
   else {
      // pop 2 operand if token is not unary, and evaluated value is pushed
      arg2 = pop_int(stack, &top);
      arg1 = pop_int(stack, &top);
      switch(token) {
         case plus:
            push_int(stack, &top, arg1+arg2); break;
         case minus:
            push_int(stack, &top, arg1-arg2); break;
         case times:
            push_int(stack, &top, arg1*arg2); break;
         case divide:
            push_int(stack, &top, arg1/arg2); break;
         case mod:
            push_int(stack, &top, arg1%arg2); break;
         default: break;
      }
   }
```

Code 9

이 프로세스가 모두 끝나면, 스택의 0번째 요소에 결과값이 저장되어 있을 것이므로, 해당 값을 리턴합니다. 만약, 프로세스가 끝났는데 스택에 2개 이상의 element들이 저장되어있다면, 받은 postfix expression이 잘못된 문법을 가지고 있는 것이므로 에러 메세지를 출력합니

다. eval_postfix() 함수의 시간복잡도는 postfix의 길이가 n일 때, $O(n)$입니다.

```
if(top != 0) { // if evaluation did not complete correctly
      printf("Invalid expression\n");
      exit(1);
   }
   return pop_int(stack, &top);
```

Code 10

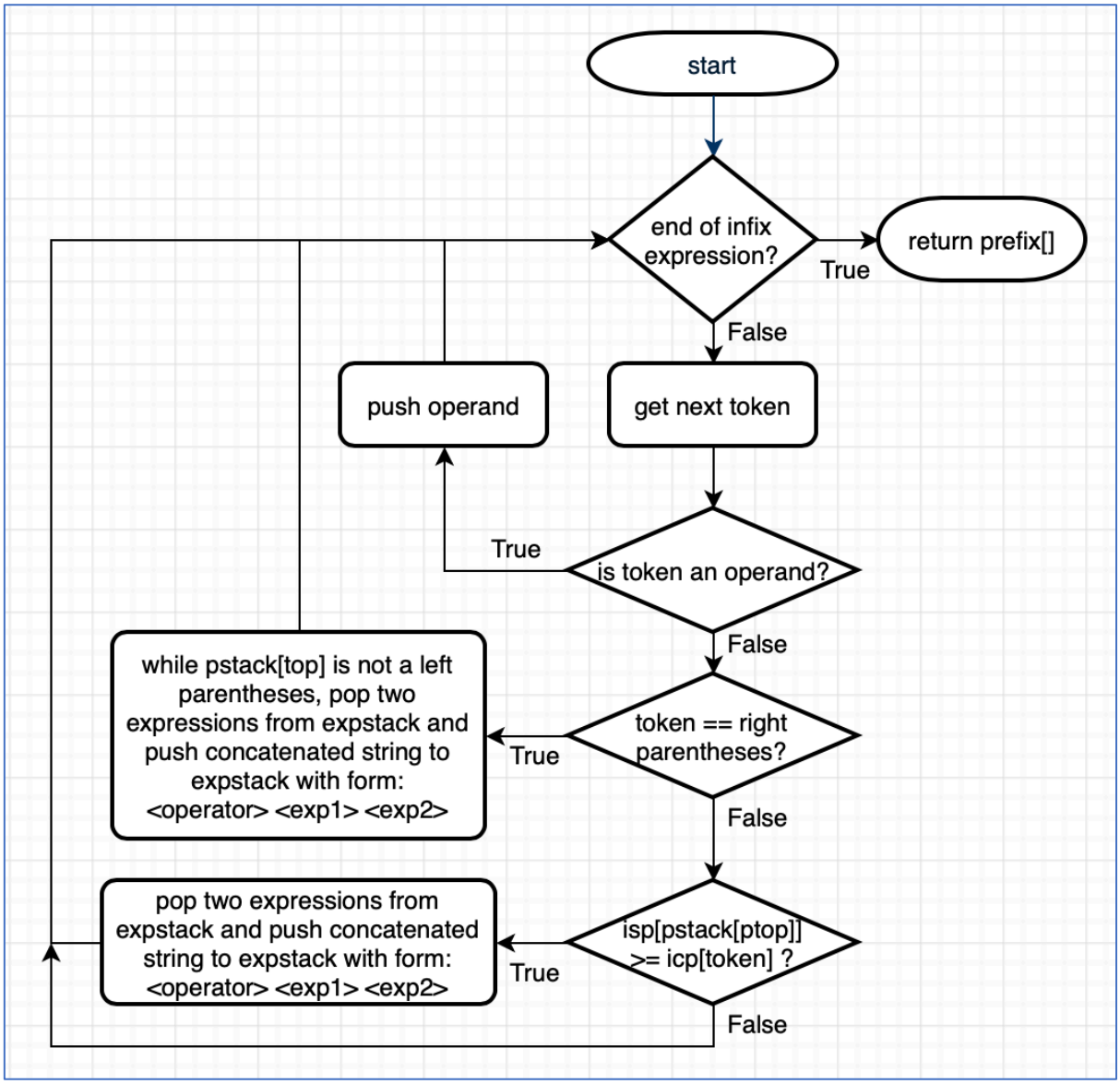
1.3 참고 사항

- to_postfix() 함수와 eval_postfix() 함수에서 사용하는 스택의 타입이 달라 pop 과 push 함수를 각각 두 개 씩 만들어야 했기 때문에, 생산적이기 못했습니다. 만약 C++, Python 과 같이 OOP 다중정의를 지원하는 언어를 활용했더라면 해당 함수를 하나로 통합시킬 수 있었을 것이므로 더 간결하고 유지보수에 유리한 프로그램을 만들 수 있었을 것입니다.
- 위에서 언급한 unary operator 를 판별하는 규칙은 오직 – operator 에서만 작동하는 것이 아닌, 모든 unary operator 에서 사용이 가능하기 때문에, minus operator 외에도 plus operator 를 지원하도록 프로그램을 다시 짜는 것은 매우 간단할 것입니다.

2. 문제2

문제2에서는 이미 알려준 infix를 prefix로 바꾸는 방법을 쓰지 않고, 다른 방법으로 바꾸는 방법을 모색하라는 조건이 있습니다. 따라서, 정석적인 방법으로 두 개의 스택을 사용해 prefix를 구하는 프로그램을 만들었습니다. 이 프로그램을 infix expression을 prefix expression으로 바꾸기 위해 함수 to_prefix()를 사용합니다.

2.1 프로그램 구조



Flowchart 3

2.2 세부 설명

이 프로그램은 expression을 담아놓기 위한 string의 배열로 이루어진 expstack, 그리고 operand를 담아놓기 위한 pstack, 총 두 가지의 스택을 사용합니다. 전달받은 infix expression의 각 요소에 대해 loop를 돌도록 구성했습니다.

```
for(int i=0 ; i<(int)strlen(infix) ; i++) {
```

```
    op = infix[i];
    token = getToken(op);
    ...
}
```

Code 11

먼저, token이 operand라면 expression을 담아놓는 expstack에 해당 문자를 push합니다. 이 때, 해당 문자는 문자열이 아닌 char 상수이기 때문에, 임시 char 배열에서 null-terminated string으로 변환해준 다음 push를 해주어야 합니다.

```
if(token == operand) {
   // push all operands
   str[0] = op;
   str[1] = '\0'; // make token into null-terminated string
   push_str(expstack, &exptop, str);
}
```

Code 12

만약, token이 right parentheses라면, pstack 안의 left parentheses를 만날 때 까지 모든 operator들을 evaluate해주어야 합니다. 이 때, expstack으로부터 두 번 pop을 해야 하며, pop한 두 expression의 앞에 pstack에서 pop한 operator를 붙여주어야 합니다. 이 때 또한 operator를 문자열로 변환해서 strcat()을 호출합니다.

```
else if(token == rightp) {
   while(ptop != -1 && pstack[ptop] != leftp) {
      // pop & push until left parentheses
      strcpy(str2, pop_str(expstack, &exptop));
      strcpy(str1, pop_str(expstack, &exptop));
      str[0] = getOp(pop(pstack, &ptop));
      str[1] = '\0'; // make token into null-terminated string

      // concat into the form: <operation> <string1> <string2>
      strcat(str, strcat(str1, str2));
      push_str(expstack, &exptop, str);
   }
   pop(pstack, &ptop);
}
```

Code 13

만약 이 중 어떠한 condition에도 속하지 않았다면, token은 right parentheses가 아닌 operator들일 것이므로, pstack의 최상위 원소와 현재 token의 precedence를 비교해 전자가 크거나 같을 경우 스택 최상위 원소를 pop하고 expstack에서 pop한 두 식과 함께 처리해 expstack에 push하고, 그렇지 않다면 현재 token을 pstack에 push합니다.

```
else {
   if(ptop != -1 && isp[pstack[ptop]] >= icp[token]) {
      // pop two expressions from stack
      strcpy(str2, pop_str(expstack, &exptop));
      strcpy(str1, pop_str(expstack, &exptop));
      str[0] = getOp(pop(pstack, &ptop));
```

```
            str[1] = '\0'; // token into null-terminated string

            // concat into the form: <operation> <string1> <string2>
            strcat(str, strcat(str1, str2));
            push_str(expstack, &exptop, str);
            push(pstack, &ptop, token);
         } else {
            push(pstack, &ptop, token);
         }
      }
```

Code 14

이 loop가 모두 끝난 뒤에, expstack과 pstack에 남아있는 원소들을 다음처럼 모두 처리해줍니다.

```
   while(exptop >= 0 && ptop >= 0) {
      // pop two expressions from stack
      strcpy(str2, pop_str(expstack, &exptop));
      strcpy(str1, pop_str(expstack, &exptop));
      str[0] = getOp(pop(pstack, &ptop));
      str[1] = '\0'; // token into null-terminated string

      // concat into the form: <operation> <string1> <string2>
      strcat(str, strcat(str1, str2));
      push_str(expstack, &exptop, str);
   }
```

Code 15

이 후, pstack 또는 expstack에 필요없는 원소가 남아있을 경우(expstack에는 1개의 원소만이 남아있어야 합니다), 식이 잘못되었다는 뜻이므로, 에러 메세지를 출력하고, 그렇지 않다면, 결과 expression을 리턴합니다. 이 함수의 시간복잡도는 infix 길이가 n일 때 $O(n)$입니다.

```
   if(exptop != 0 || ptop != -1) {
      // check if the process did not end correctly
      printf("Invalid expression\n");
      exit(1);
   }
   // copy to prefix string.
   strcpy(prefix, expstack[exptop]);
   return prefix;
```

Code 16

2.3 참고 사항

- 이 프로그램에서는 expression을 담기 위한 스택의 크기를 처음부터 정적배열로 선언했습니다. 만약 동적할당을 통해 스택의 크기를 pop, push에 따라 조정했다면, 메모리를 현저히 적게 사용할 수 있도록 최적화가 가능했을 것입니다. OOP 언어를 사용한다면 이 과정을 구현하는 것 또한 간단하게 이룰 수 있습니다.
- Infix를 반전시키고, 그것을 postfix로 변환한 다음 다시 반전시켜 prefix를 얻는 알고리즘과 비교한다면, 이 알고리즘은 문자열 스택을 추가로 더 사용하기에 공간을 더 많이 쓰

게 됩니다. 따라서 메모리 용량이 극도로 중요한 상황에서는 이 알고리즘보다는 postfix 를 사용한 알고리즘이 더 효과적입니다.

3. 문제 코드

3.1 문제1 코드

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef enum {
   leftp, rightp, plus, minus, times, divide, mod, unary, operand
} precedence;
/**
 * Pops precedence from stack. If stack is empty, prints error and terminates
 * program.
 * @param  stack Array for the stack that will be popped
 * @param  top   Pointer to integer of stack top index (-1 for empty stack)
 * @return       Popped value
 */
precedence pop(precedence* stack, int* top) {
   precedence value;
   if(*top <= -1) { //popping empty stack
      printf("Invalid expression\n");
      exit(1);
   }
   value = stack[(*top)--];
   return value;
}
/**
 * Same functionality as pop(), but with integers
 * @param  stack Array for the stack that will be popped
 * @param  top   Pointer to integer of stack top index (-1 for empty stack)
 * @return       Popped value
 */
char pop_int(int* stack, int* top) {
   int value;
   if(*top <= -1) { //popping empty stack
      printf("Invalid expression\n");
      exit(1);
   }
   value = stack[(*top)--];
   return value;
}
/**
 * Push precedence onto stack.
 * @param stack Array for the stack that is used for this procedure.
 * @param top   Pointer to integer value for stack top index
 * @param token Procedence value that needs to be pushed.
 */
void push(precedence* stack, int* top, precedence token) {
   stack[++(*top)] = token; // push token into stack
}
/**
 * Same functionality as push(), but with integers.
```

```
 * @param stack Array for the stack that is used for this procedure.
 * @param top   Pointer to integer value for stack top index
 * @param token Procedence value that needs to be pushed.
 */
void push_int(int* stack, int* top, int token) {
   stack[++(*top)] = token; // push token into stack
}
/**
 * Get corresponding character for given precedence token
 * @param  token Precedence value that you wish to get character for
 * @return       Character for given precedence.
 */
char getOp(precedence token) {
   switch(token) {
      case leftp: return '('; break;
      case rightp: return ')'; break;
      case plus: return '+'; break;
      case minus: return '-'; break;
      case times: return '*'; break;
      case divide: return '/'; break;
      case mod: return '%'; break;
      case unary: return '#'; break;
      default: return -1;
   }
}
/**
 * Does the opposite of getOp. Gets precedence value from given operation char.
 * @param  op Character that needs analysising.
 * @return    Corresponding precedence value for op.
 */
precedence getToken(char op) {
   switch(op) {
      case '(': return leftp; break;
      case ')': return rightp; break;
      case '+': return plus; break;
      case '-': return minus; break;
      case '*': return times; break;
      case '/': return divide; break;
      case '%': return mod; break;
      case '#': return unary; break;
      default: return operand; break;
   }
}
/**
 * Evaluate postfix expressions
 * @param  expression Postfix expression to evaluate
 * @return            Result in integer.
 */
int eval_postfix(char* expression) {
   int stack[20];
   int top = -1;
```

```
   int result;
   int op, arg1, arg2;
   precedence token;

   for(int i=0 ; i<(int)strlen(expression) ; i++) {
      op = expression[i];
      token = getToken(op);
      if(token == operand) {
         // push all operands into stack
         push_int(stack, &top, op-'0');
      } else if(token == unary) {
         // pop 1 operand if token is unary. evaluated value is pushed
         arg2 = pop_int(stack, &top);
         push_int(stack, &top, -arg2);
      } else {
         // pop 2 operand if token is not unary, and evaluated value is pushed
         arg2 = pop_int(stack, &top);
         arg1 = pop_int(stack, &top);
         switch(token) {
            case plus:
               push_int(stack, &top, arg1+arg2); break;
            case minus:
               push_int(stack, &top, arg1-arg2); break;
            case times:
               push_int(stack, &top, arg1*arg2); break;
            case divide:
               push_int(stack, &top, arg1/arg2); break;
            case mod:
               push_int(stack, &top, arg1%arg2); break;
            default: break;
         }
      }
   }
   if(top != 0) { // if evaluation did not complete correctly
      printf("Invalid expression\n");
      exit(1);
   }
   return pop_int(stack, &top);
}
/**
 * Translate infix expression to postfix expression.
 *    Unary operations are included, and are marked as '#' in resulting
 *    expression. Minus operators are identified as unary if it matches
 *    any of the following conditions:
 *    - the operator is the first thing in the expression.
 *    - the operator comes after left parentheses.
 *
 * @param  infix   Infix expression to translate
 * @param  postfix Character array where resulting postfix expressions will be
 *                 written to.
 * @return         String of the postfix expression.
```

```
 */
char* to_postfix(char* infix, char* postfix) {
   precedence stack[20];
   char op;
   int top = -1, post_idx = 0, i;
   int f_unary = 0; // if this is set, next minus operation is unary.
   precedence token;
   // isp and icp order: leftp, rightp, plus, minus, times, divide, mod, unary
   // operand is missing in these arrays since it shouldn't really be a thing
   int isp[8] = {0, 9, 2, 2, 3, 3, 3, 4};
   int icp[8] = {10, 9, 2, 2, 3, 3, 3, 4};
   for(i=0 ; i<(int)strlen(infix) ; i++) {
      op = infix[i];
      token = getToken(op);
      if (token == minus) { // minus unary check
         if(i == 0) {
            f_unary = 1;
            continue;
         } else if(getToken(infix[i-1]) != operand &&
               getToken(infix[i-1]) != rightp) {
            f_unary = 1;
            continue;
         }
      }
      if(token == operand) {
         postfix[post_idx++] = op;
         if(f_unary) { //unary print process
            postfix[post_idx++] = getOp(unary);
            f_unary = 0;
         }
      } else if(token == rightp) { // token is right parentheses
         while(stack[top] != leftp) { //check for top validity
            // pop until left parentheses
            postfix[post_idx++] = getOp(pop(stack, &top));
         }
         pop(stack, &top); // pop left parentheses from stack
         if(f_unary) { //unary print process
            postfix[post_idx++] = getOp(unary);
            f_unary = 0;
         }
      } else {
         while(top != -1 && isp[stack[top]] >= icp[token]) {
            // while in-stack-precedence of stack's top
            //       >= incoming precedence of token
            postfix[post_idx++] = getOp(pop(stack, &top)); // pop stack
         }
         push(stack, &top, token);
      }
   }
   if(i != (int)strlen(infix)) {
      //error occurred
```

```
      return NULL;
   }
   while(top != -1) {
      // pop remaining operators
      postfix[post_idx++] = getOp(pop(stack, &top));
   }
   postfix[post_idx] = '\0'; // for null-terminating string
   return postfix;
}
int main(int argc, const char* argv[]) {
   char infix[21], postfix[21];
   int result;

   printf("Input: ");
   scanf("%s", infix);

   if(!to_postfix(infix, postfix)) { //convert into postfix
      printf("Invalid expression\n");
      return 0;
   }
   result = eval_postfix(postfix); // evaluate postfix expression generated.

   printf("Postfix: %s\n", postfix);
   printf("Result: %d\n", result);
   return 0;
}
```

3.2 문제2 코드

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef enum {
   leftp, rightp, plus, minus, times, divide, mod, operand
} precedence;
/**
 * Pops precedence from stack. If stack is empty, prints error and terminates
 * program.
 * @param  stack Array for the stack that will be popped
 * @param  top   Integer of stack top index (-1 for empty stack)
 * @return       Popped value
 */
precedence pop(precedence* stack, int* top) {
   precedence value;
   if(*top <= -1) { //popping empty stack
      printf("Invalid expression\n");
      exit(1);
   }
   value = stack[(*top)--];
   return value;
```

```
}
/**
 * Push precedence onto stack.
 * @param stack Array for the stack that is used for this procedure.
 * @param top   Pointer to integer value for stack top index
 * @param token Procedence value that needs to be pushed.
 */
void push(precedence* stack, int* top, precedence token) {
   stack[++(*top)] = token; // push token into stack
}
/**
 * Same functionality as pop(), but the stack consists of strings.
 * @param stack[][] 2-D character array for storing strings as stack.
 * @param top       Pointer to integer representing top index of the stack
 */
char* pop_str(char stack[20][21], int* top) {
   if(*top <= -1) { //popping empty stack
      printf("Invalid expression\n");
      exit(1);
   }
   return stack[(*top)--];
}
/**
 * Same functionality as push(), but the stack consists of strings.
 * @param stack[][]  2-D character array for storing strings as stack.
 * @param top       Pointer to integer representing top index of the stack
 * @param expression Null-terminated string to push into the stack.
 */
void push_str(char stack[20][21], int* top, char* expression) {
   strcpy(stack[++(*top)], expression); // since token is null-terminated string
}
/**
 * Get corresponding character for given precedence token
 * @param  token Precedence value that you wish to get character for
 * @return       Character for given precedence.
 */
char getOp(precedence token) {
   switch(token) {
      case leftp: return '('; break;
      case rightp: return ')'; break;
      case plus: return '+'; break;
      case minus: return '-'; break;
      case times: return '*'; break;
      case divide: return '/'; break;
      case mod: return '%'; break;
      default: return -1;
   }
}
/**
 * Does the opposite of getOp. Gets precedence value from given operation char.
 * @param  op Character that needs analysising.
```

```
 * @return    Corresponding precedence value for op.
 */
precedence getToken(char op) {
   switch(op) {
      case '(': return leftp; break;
      case ')': return rightp; break;
      case '+': return plus; break;
      case '-': return minus; break;
      case '*': return times; break;
      case '/': return divide; break;
      case '%': return mod; break;
      default: return operand; break;
   }
}
/**
 * Translate infix expression to prefix expression. This is done by utilising
 * two stacks, one for operators, and one for expressions(+operands).
 * @param  infix  String containing infix expression to translate.
 * @param  prefix Character array where resulting prefix expressions will be
 *                written to.
 * @return        Translated prefix string.
 */
char* to_prefix(char* infix, char* prefix) {
   precedence pstack[20]; //precedence(operator) stack
   precedence token;
   char expstack[20][21]; //expression(operand) stack
   int ptop = -1, exptop = -1;
   char str[21], str1[21], str2[21], op;
   // isp and icp order: leftp, rightp, plus, minus, times, divide, mod
   // operand is missing in these arrays since it shouldn't really be a thing
   int isp[7] = {0, 9, 2, 2, 3, 3, 3};
   int icp[7] = {10, 9, 2, 2, 3, 3, 3};

   for(int i=0 ; i<(int)strlen(infix) ; i++) {
      op = infix[i];
      token = getToken(op);
      if(token == operand) {
         // push all operands
         str[0] = op;
         str[1] = '\0'; // make token into null-terminated string
         push_str(expstack, &exptop, str);
      } else if(token == rightp) {
         while(ptop != -1 && pstack[ptop] != leftp) {
            // pop & push until left parentheses
            strcpy(str2, pop_str(expstack, &exptop));
            strcpy(str1, pop_str(expstack, &exptop));
            str[0] = getOp(pop(pstack, &ptop));
            str[1] = '\0'; // make token into null-terminated string

            // concat into the form: <operation> <string1> <string2>
            strcat(str, strcat(str1, str2));
```

```
            push_str(expstack, &exptop, str);
         }
         pop(pstack, &ptop);
      } else {
         if(ptop != -1 && isp[pstack[ptop]] >= icp[token]) {
            // pop two expressions from stack
            strcpy(str2, pop_str(expstack, &exptop));
            strcpy(str1, pop_str(expstack, &exptop));
            str[0] = getOp(pop(pstack, &ptop));
            str[1] = '\0'; // token into null-terminated string

            // concat into the form: <operation> <string1> <string2>
            strcat(str, strcat(str1, str2));
            push_str(expstack, &exptop, str);
            push(pstack, &ptop, token);
         } else {
            push(pstack, &ptop, token);
         }
      }
   }
   while(exptop >= 0 && ptop >= 0) {
      // pop two expressions from stack
      strcpy(str2, pop_str(expstack, &exptop));
      strcpy(str1, pop_str(expstack, &exptop));
      str[0] = getOp(pop(pstack, &ptop));
      str[1] = '\0'; // token into null-terminated string

      // concat into the form: <operation> <string1> <string2>
      strcat(str, strcat(str1, str2));
      push_str(expstack, &exptop, str);
   }
   if(exptop != 0 || ptop != -1) {
      // check if the process did not end correctly
      printf("Invalid expression\n");
      exit(1);
   }
   // copy to prefix string.
   strcpy(prefix, expstack[exptop]);
   return prefix;
}
int main(int argc, const char* argv[]) {
   char infix[21], prefix[21];

   printf("Infix: ");
   scanf("%s", infix);

   if(!to_prefix(infix, prefix)) { // translate infix to prefix
      printf("Invalid expression\n");
      return 0;
   }
   printf("Prefix: %s\n", prefix);
```

```
    return 0;
}
```