컴퓨터학부 20162448 김병준

Jamzom\_jago\_sheepda

1. **개요**

수많은 세기를 거쳐오며 사람들은 복잡한 작업에 대해서 사람이 직접 해오던 일을 기계에게 맡기기 시작하였다. 그리고 정보화를 거치면서 자동화되었다. 그 중에는 학생들이 본 시험의 답안에 대해서 채점을 하는 작업도 포함할 수 있다. 대표적인 예시가 바로 광학마크인식(OMR)이다. 학생들이 답안을 용지에 컴퓨터가 인식할 수 있는 펜으로 작성을 하면, 그 작성된 답안을 컴퓨터가 인식하여 답안 테이블을 구성하고 채점을 한다. 본 과제는 학생들이 제출한 텍스트 파일과 프로그램 파일 답안에 대하여 텍스트 비교 및 프로그램 컴파일 및 실행 결과 비교를 통하여 학생들의 답안을 자동화하여 채점하는 프로그램이다.

이 프로그램은 프로그램 실행 전, 문제별로 점수를 할당할 수 있으며, 문제 별 점수를 테이블화 하여 파일에 기록한다. 그리고 프로그램 문제의 경우 스레드 옵션이 주어져야 할 경우에 대한 처리도 제공한다. 또한 프로그램 실행 후 학생들의 채점 결과를 각 문제별로 볼 수 있는 파일도 제공한다. 그리고 테이블이 생성되면 필요한 경우 해당 학생들에 대한 오답 목록을 출력한다.

1. **분석**

이 프로그램이 채점이란 동작을 진행하기 위해서는 표본이 필요하다. 그것은 학생들이 제출한 답안과 그것이 정답과 오답의 기준이 될 수 있는 답안이다. 우리는 이러한 답안들을 묶을 수 있는 특성이 필요한데 이것을 디렉토리로 묶어보고자 한다.

먼저 표준이 되는 답안의 경우 문제의 번호, 개수, 파일 유형 등 특성 세개가 반드시 필요하다. 이것은 채점을 올바른 문제에 올바른 횟수로 빈칸문제인지 프로그램 문제인지를 구분할 수 있는 지표가 되기 때문이다. 그리고 이러한 답안 파일들을 가지고 있는 디렉토리의 경로가 반드시 필요하다. 이 경로를 우리는 명령을 실행할 때 인자로 학생, 정답 답안 디렉토리 명을 넘겨주기 때문에 해당 디렉토리 경로를 알 수 있다. 그리고 이러한 인자로 받아온 정답 답안 경로를 반드시 기억해 둘 필요가 있다.

반대로 이 프로그램의 궁극적인 목표라고 할 수 있는 학생의 답안의 경우이다. 표준 답안과 마찬가지로 문제의 번호, 개수, 파일 유형 총 세개의 특성이 반드시 필요하다. 그러나 이러한 특성들을 다시 학생들을 구분 짓는 학번으로 우리는 묶을 필요가 있다. 그렇지 않으면 우리는 답안이 누구의 답안인지 알 수 없기 때문이다. 그러므로 학생 답안의 경로 중에는 반드시 학번이 들어간 중간 경로가 필요하다.

**2-1. ssu\_score.h**

해당 헤더파일에는 main.c, ssu\_score.c에서 사용하는 함수들의 프로토타입과 전역변수들이 제시되어 있다.

**2-2. main.c**

명세서에서 제공된 구현해야 하는 기능 중 하나는 프로그램 실행시간을 측정 후 출력하는 것이다. 따라서 메인 프로그램은 서브 프로그램(채점 프로그램)이 실행 완료되는데 걸리는 시간을 측정 후 출력한다. 서브프로그램의 실행시간을 측정하는 메인 프로그램의 전체적인 흐름도는 [그림 1]과 같다.



[그림 1] 메인 프로그램 흐름도

**2-2-1. int main(int argc, char \*argv[])**

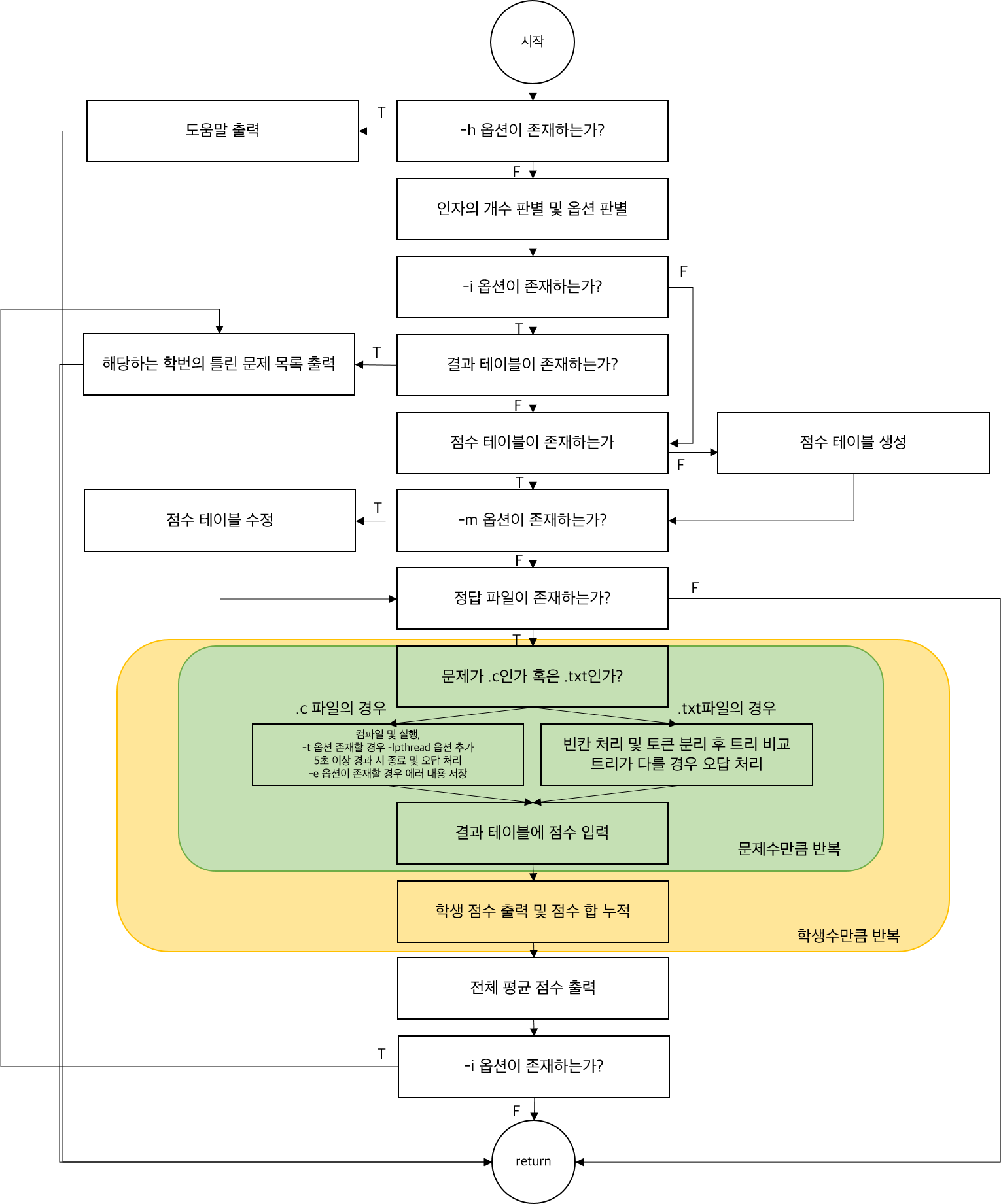
서브 프로그램 호출과 더불어 시작 및 종료시간 측정기능을 제공함.

**2-2-2. void ssu\_runtime(struct timeval \*begin\_t, struct timeval \*end\_t)**

메인 함수에서 기록한 시작시간(begin\_t)과 종료시간(end\_t)를 사용하여 실행 시간을 계산하고 화면에 출력한다. 두 시간의 차를 계산할 때 gettimeofday() 함수를 통해 저장된 시간들이 몫과 나머지처럼 딱 잘라지는데, 이때 시작시간이 종료시간보다 마이크로초가 더 큰 경우가 존재한다.(ex. 3분 13초 – 1분 42초) 따라서 이 경우를 조건문을 통해 예외처리를 해주어서 초 변환을 시켜주어서 해결하였다.

**2-3. ssu\_score.c**

서브 프로그램(채점 프로그램)은 정답 답안을 기준으로 학생들이 제출한 답안(빈칸, 프로그램)을 채점 후 결과를 파일로 기록 및 출력 후 종료한다. 해당 프로그 서브 프로그램이 채점 후 프로그램 종료까지의 전체적인 흐름도는 [그림 2]와 같다



[그림 2] 서브 프로그램 흐름도

서브 프로그램은 함수 호출 시, 메인 프로그램으로부터 넘겨받은 argc, argv인자를 활용하여 인자에 포함되어 있는 옵션과 개수를 판별한다. 그 전에 만약 -h옵션이 존재하면 사용법만 출력하고 바로 종료한다.

-i옵션이 존재할 경우 결과 테이블의 유무에 따라서 프로그램의 종료 시점이 달라진다. 결과 테이블이 존재할 경우 바로 테이블을 참조하여 인자로 받은 학번의 틀린 문제 목록을 출력하고 종료한다. 만약 존재하지 않을 경우에는 점수 테이블을 생성한 후 -m옵션을 확인한다.

-m옵션이 존재할 경우 점수 테이블을 수정하고, 없으면 정답파일을 확인한다. 파일을 확인하면서 c파일과 txt파일 경우로 나뉘게 된다.

c파일의 경우 컴파일 및 실행을 하며 -t옵션이 주어진 문제에 대해선 gcc 실행시 -lpthread 옵션을 주게 된다. 이후 실행하였을 때 5초이상 런타임이 지속될 경우 오답처리를 하고, -e옵션이 주어졌을 경우 컴파일 에러시 에러 내용을 인자로 준 디렉토리에 학번별로 저장된다.

txt파일의 경우 답안 내용에 대해 빈칸을 없애고 토큰화 시킨 뒤 트리생성 후 학생 답안과 정답 답안의 내용을 비교한다. 트리가 서로 다를 경우 오답처리한다.

이후 결과테이블에 점수를 입력하고 이것은 학생 한명당 문제 수만큼 반복된다. 이후 모든 문제를 채점하였으면 학생의 점수를 화면에 출력하고, 학생의 총점을 점수합에 더한다.

모든 학생들의 채점이 끝나면 학생들의 전체 평균 점수를 출력하고 다시한번 -i옵션을 체크한 뒤 서브프로그램을 종료한다.

**2-3-1. 전역 변수**

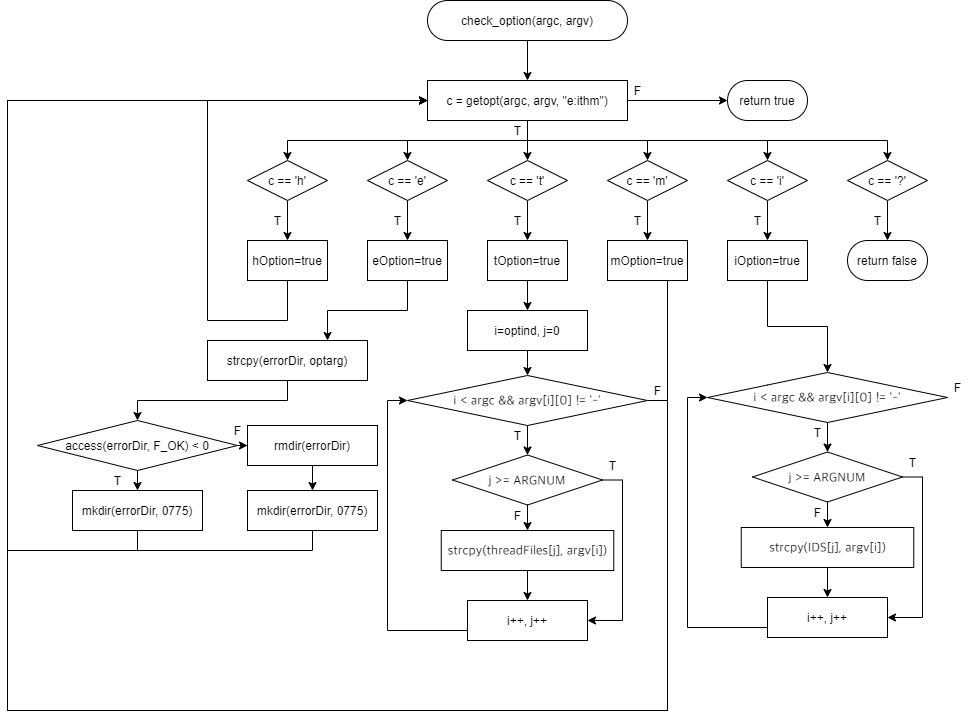
서브 프로그램에서 채점을 하기 위해 공통적으로 필요한 데이터들을 선언한다. 주로 문제 및 할당 점수를 가지고 있는 점수 테이블 및 학번 테이블 구조체와 현재 절대경로, 학생 답안 경로, 정답 답안 경로, 에러 저장 디렉토리 경로, -lpthread를 줄 문제 번호들을 담고 있는 배열, 옵션 확인 변수를 선언한다.

**2-3-2. void ssu\_socre(int argc, char \*argv[])**

서브 프로그램에서 메인 함수의 기능을 담당하고 있는 함수로서, 서브 프로그램의 흐름도를 대표하는 함수이다. 기존 프로그램에서 -c옵션의 조건 탐색 시점을 활용하여 -i옵션을 구현하였고, -m옵션의 조건 탐색 시점을 점수 테이블 생성 후로 작성하여 옵션 호출 시점을 구현하였다.

**2-3-3. int check\_option(int argc, char \*argv[])**

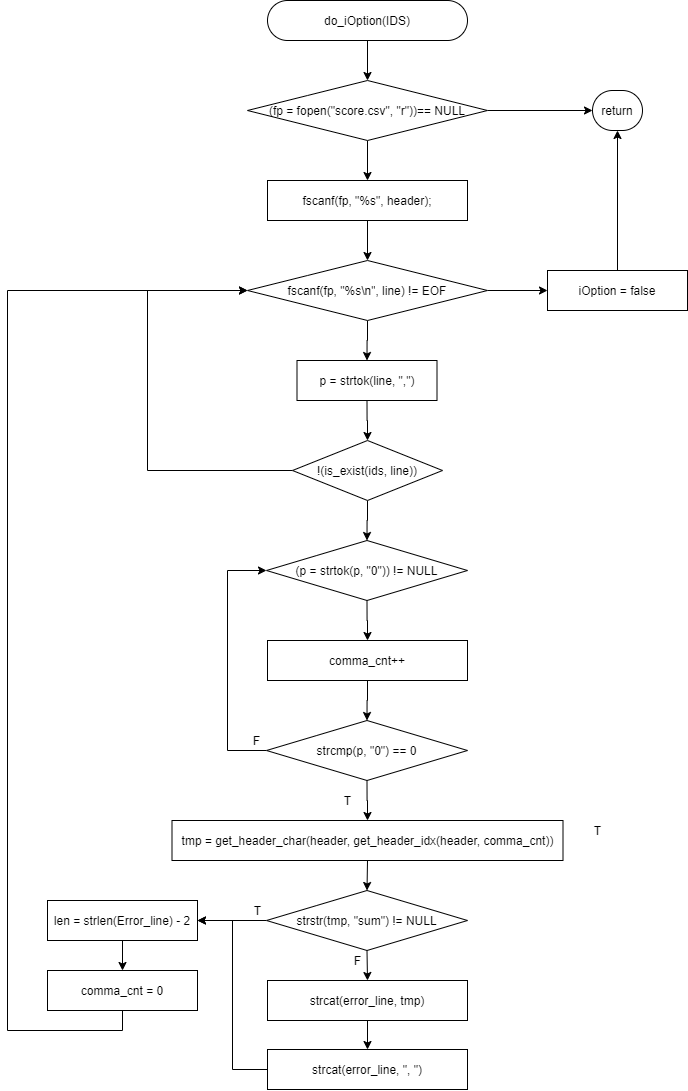
이 함수는 getopt()함수를 사용하여 명령어에 주어진 인자들을 읽고, 필요한 인자 내용을 저장해주는 함수이며, 프로그램의 흐름도는 [그림 3]과 같다.



[그림 3] check\_option() 흐름도

**2-3-4. void do\_iOption(char (\*ids)[FILELEN])**

이 함수는 -i옵션과 인자로 학번이 주어졌을 경우, 해당 학번이 존재할 때 틀린 문제에 대하여 목록을 출력하여 준다. 이 함수의 흐름도는 [그림 4]와 같다. 이 함수는 결과테이블이 존재하지 않을 경우 아무것도 하지 않고 반환된다. 그러나 존재할 경우 주어진 인자에 대한 처리를 진행하게 된다. 인자로 주어진 학번에 대하여 결과 테이블을 탐색하고, 오답 처리된 문제(0점)에 대하여 발견 되기까지의 쉼표(,)의 개수를 카운트한다. 해당 쉼표의 개수로 문제 파일명이 위치하는 인덱스의 시작과 문자열을 받아온 후 출력한다.



[그림 4] do\_iOption() 흐름도

**2-3-5. int get\_header\_idx(char \*header, int comma\_cnt)**

이 함수는 인자로 주어진 문자열 header와 쉼표 개수 인자 comma\_cnt를 이용하여 header 문자열에서 comma\_cnt만큼 쉼표가 카운트 될 때 까지 인덱스를 이동한 뒤, 그 다음 인덱스를 반환한다.

**2-3-6. char \*get\_header\_char(char \*header, int idx)**

이 함수는 인자로 주어진 문자열 header와 인덱스 인자 idx를 이용하여 header문자열에서 idx만큼 이동 후 다음 쉼표(,)가 나오기 전까지의 문자들을 반환한다.

**2-3-7. int is\_exist(char (\*src)[FILELEN], char \*target)**

**2-3-8. void set\_scoreTable(char \*ansDir)**

**2-3-9. void read\_scoreTable(char \*path)**

**2-3-10. void make\_scoreTable(char \*ansDir)**

**2-3-11. void write\_scoreTable(char \*filename)**

**2-3-12. void set\_idTable(char \*stuDir)**

**2-3-13. void sort\_idTable(int size)**

**2-3-14. void sort\_scoreTable(int size)**

**2-3-15. void get\_qname\_number(char \*qname, int \*num1, int \*num2)**

**2-3-16. int get\_create\_type()**

**2-3-17. void score\_students()**

**2-3-18. double score\_student(int fd, char \*id)**

**2-3-19. void write\_first\_row(int fd)**

**2-3-20. char \*get\_answer(int fd, char \*result)**

**2-3-21. int score\_blank(char \*id, char \*filename)**

**2-3-22. double score\_program(char \*id, char \*filename)**

**2-3-23. int is\_thread(char \*qname)**

**2-3-24. double compile\_program(char \*id, char \*filename)**

**2-3-25. double check\_error\_warning(char \*filename)**

**2-3-26. int execute\_program(char \*id, char \*filename)**

**2-3-27. pid\_t inBackground(char \*name)**

**2-3-28. int compare\_resultfile(char \*file1, char \*file2)**

**2-3-29. void redirection(char \*command, int new, int old)**

**2-3-30. int get\_file\_type(char \*filename)**

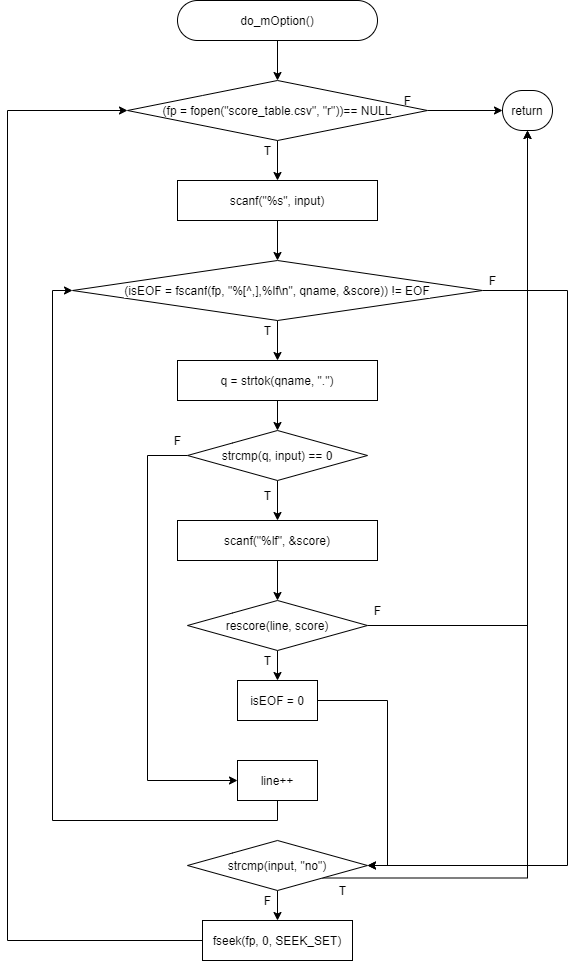
**2-3-31. void rmdirs(const char \*path)**

**2-3-32. void to\_lower\_case(char \*c)**

**2-3-33. void print\_usage()**

**2-3-34. void do\_mOption()**

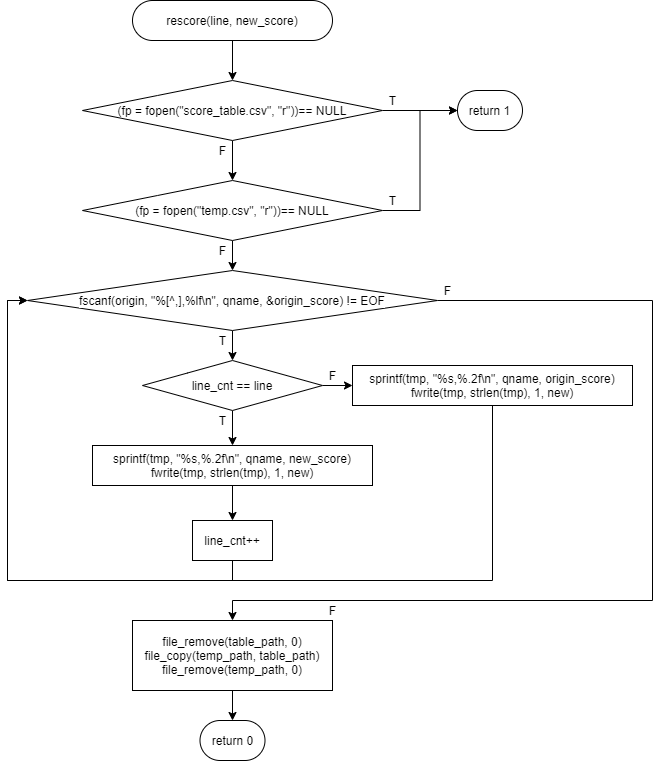
이 함수는 -m옵션이 주어졌을 경우에 점수 테이블의 문제 할당 점수를 수정 및 저장해준다. 먼저 점수 테이블이 존재하지 않을경우 함수를 반환한다. 만약 점수 테이블이 존재할 경우, 문제 번호를 입력받는다. 문제번호가 점수 테이블에 존재할 경우 해당 라인 번호를 기억한 다음, 변경할 새로운 점수를 입력받는다. 입력받은 점수와 라인 번호를 토대로 기존 점수테이블을 복사하다가, 해당 라인에 접근하게 되면 점수 부분을 새로 입력받은 점수로 작성한다. 그 이후 다시 기존의 점수 테이블을 복사한다. 이 작업은 반복 가능하며, 문제번호 입력에 “no”를 입력했을 경우 이 함수는 반환된다.



[그림 5] do\_mOption() 흐름도

**2-3-35. int rescore(int line, double new\_score)**

-m 옵션이 주어졌을 경우, 수정할 문제의 라인의 인덱스인 line과 갱신될 점수 new\_score 변수를 이용하여 변경된 점수가 반영된 score\_table.csv를 생성한다. 이 함수의 흐름도는 [그림 6]과 같다.



[그림 6] rescore() 흐름도

**2-3-36. void file\_remove(const char \*file, \_Bool isDir)**

주어진 파일 및 디렉토리(경로)를 삭제해주는 함수이다.

**2-3-37. void file\_copy(const char \*from, const char \*to)**

주어진 파일을 복사해주는 함수이다.

**2-4. blank.h**

**2-5. blank.c**

서브 프로그램의 기능 중 빈칸 문제 채점에 관련된 함수들이 작성되어있다. 빈칸 문제 채점을 위해서 주어진 답안의 문자열에 대하여 파싱을 진행하게 된다. 파싱을 하기 위해선 먼저 주어진 문자열의 토큰화(make\_tokens)시킨다, 생성된 토큰들을 바탕으로 트리를 구축(make\_trees)한다. 이때 트리는 연산자 우선순위에 의해서 높은 순서에서 낮은 순서로 재귀적으로 생성된다. 그리고 높은 우선순위의 연산자의 좌우 토큰들을 비교해서 그 토큰 안에 낮은 순서의 연산자나 괄호가 있을 경우, 그에 대한 함수를 재귀적으로 호출하게 된다. 그리고 이렇게 생성된 학생답안과 정답답안의 트리들을 비교(compare\_tree)한다.

**2-5-1. 전역 변수**

blank.c 안에서 사용될 데이터타입이 담긴 배열과 연산자 우선순위가 선언되어있다.

**2-5-2. void compare\_tree(node \*root1, node \*root2, int \*result)**

|  |
| --- |
| void compare\_tree(node \*root1, node \*root2, int \*result) // std\_root, ans\_root 파스트리 비교, 같음:1, 다름:0  {  node \*tmp;  int cnt1, cnt2;  if(root1 == NULL || root2 == NULL){ // 비교할 트리 중 하나라도 트리가 존재하지 않을 경우  \*result = false;  return;  }  if(!strcmp(root1->name, "<") || !strcmp(root1->name, ">") || !strcmp(root1->name, "<=") || !strcmp(root1->name, ">=")) {  if(strcmp(root1->name, root2->name) != 0){ // 학생 문자와 정답 문자가 같지 않을 경우  if(!strncmp(root2->name, "<", 1)) // 정답 문자가 '<'  strncpy(root2->name, ">", 1); // 정답 문자에 '>' 복사  else if(!strncmp(root2->name, ">", 1)) // 정답 문자가 '>'  strncpy(root2->name, "<", 1); // 정답 문자에 '<' 복사  else if(!strncmp(root2->name, "<=", 2)) // 정답 문자가 "<="  strncpy(root2->name, ">=", 2); // 정답 문자에 ">=" 복사  else if(!strncmp(root2->name, ">=", 2)) // 정답 문자가 ">="  strncpy(root2->name, "<=", 2); // 정답 문자에 "<=" 복사  root2 = change\_sibling(root2); // 다음 형제 노드 이동  }  }  if(strcmp(root1->name, root2->name) != 0){ // 학생 문자와 정답 문자가 같지 않을 경우  \*result = false;  return;  }  if((root1->child\_head != NULL && root2->child\_head == NULL)  || (root1->child\_head == NULL && root2->child\_head != NULL)){ // 비교할 자식 노드가 하나라도 존재하지 않을 경우  \*result = false;  return;  }  else if(root1->child\_head != NULL){ // 학생 답안의 자식이 존재 할 경우  if(get\_sibling\_cnt(root1->child\_head) != get\_sibling\_cnt(root2->child\_head)){ // 형제 개수가 같지 않으면  \*result = false;  return;  }  if(!strcmp(root1->name, "==") || !strcmp(root1->name, "!="))  {  compare\_tree(root1->child\_head, root2->child\_head, result); // 학생 답안과 정답 답안 트리 비교(재귀)  if(\*result == false) // 여러개의 답의 경우, 정답 답안 형제 트리 순환 비교  {  \*result = true;  root2 = change\_sibling(root2);  compare\_tree(root1->child\_head, root2->child\_head, result);  }  }  else if(!strcmp(root1->name, "+") || !strcmp(root1->name, "\*")  || !strcmp(root1->name, "|") || !strcmp(root1->name, "&")  || !strcmp(root1->name, "||") || !strcmp(root1->name, "&&"))  {  if(get\_sibling\_cnt(root1->child\_head) != get\_sibling\_cnt(root2->child\_head)){ // 형제 개수가 같지 않으면  \*result = false;  return;  }  tmp = root2->child\_head;  while(tmp->prev != NULL) // 정답의 이전 형제가 존재할 경우  tmp = tmp->prev;  while(tmp != NULL) // 여러개의 정답 트리 중 처음부터 학생 트리와 순회 비교  {  compare\_tree(root1->child\_head, tmp, result);    if(\*result == true)  break;  else{  if(tmp->next != NULL)  \*result = true;  tmp = tmp->next;  }  }  }  else compare\_tree(root1->child\_head, root2->child\_head, result);  }  if(root1->next != NULL){ // 학생 답안 트리의 형제가 존재하지 않을 경우  if(get\_sibling\_cnt(root1) != get\_sibling\_cnt(root2)){ // 형제의 개수가 같지 않을 경우  \*result = false;  return;  }  if(\*result == true) // 답안이 재귀로 다시 들어오면  {  tmp = get\_operator(root1); // 학생 답안의 노드 부모 참조    if(!strcmp(tmp->name, "+") || !strcmp(tmp->name, "\*") // 부모노드가 연산자일 경우  || !strcmp(tmp->name, "|") || !strcmp(tmp->name, "&")  || !strcmp(tmp->name, "||") || !strcmp(tmp->name, "&&"))  {  tmp = root2;    while(tmp->prev != NULL) // 형제노드가 존재할 경우  tmp = tmp->prev;  while(tmp != NULL) // 형제노드가 null이 아닐경우  {  compare\_tree(root1->next, tmp, result); // 형제들과 비교  if(\*result == true)  break;  else{  if(tmp->next != NULL)  \*result = true;  tmp = tmp->next;  }  }  }  else  compare\_tree(root1->next, root2->next, result);  }  }  } |

**2-5-3. int make\_tokens(char \*str, char tokens[TOKEN\_CNT][MINLEN])**

|  |
| --- |
| int make\_tokens(char \*str, char tokens[TOKEN\_CNT][MINLEN]) // 주어진 문자열의 토큰화, TOKEN\_CNT:50, MINLEN:64  {  char \*start, \*end;  char tmp[BUFLEN];  char str2[BUFLEN];  char \*op = "(),;><=!|&^/+-\*\"";  int row = 0;  int i;  int isPointer;  int lcount, rcount;  int p\_str;    clear\_tokens(tokens); // 토큰 초기화  start = str; // 정답 문자열 시작 포인터 지정    /\*  if(is\_typeStatement(str) == 0) // 잘못된 작성일 경우, gcc 체크  return false;  \*/  while(1)  {  // strpbrk(): 두번째 인자 문자열의 문자들 중 하나라도 첫번째 문자열에 존재한다면 해당 문자가 위치한 두번째의 주소 리턴  if((end = strpbrk(start, op)) == NULL) // start에서 op의 문자들 중 하나라도 없다면 break  break;  if(start == end){ // 시작과 끝의 문자가 같다면    if(!strncmp(start, "--", 2) || !strncmp(start, "++", 2)) { // 앞에 두글자가 전위 증감연산자일 경우  if(!strncmp(start, "++++", 4)||!strncmp(start,"----",4)) // 잘못된 증감연산자 사용  return false;  if(is\_character(\*ltrim(start + 2))){ // 왼쪽 공백 제거 후, 전위 증감연산일 때 문자가 왔다면  if(row > 0 && is\_character(tokens[row - 1][strlen(tokens[row - 1]) - 1])) // ex. abc-- abc;와 같은 경우  return false;  end = strpbrk(start + 2, op); // 다음 연산자까지 토큰 탐색  if(end == NULL) // 토큰의 끝을 탐색하지 못했을 경우  end = &str[strlen(str)];  while(start < end) { // 현재 연산자부터 다음 연산자까지  if(\*(start - 1) == ' ' && is\_character(tokens[row][strlen(tokens[row]) - 1])) // start 전 문자가 공백이고 현재 토큰의 마지막 인자가 문자일 경우  return false;  else if(\*start != ' ') // 공백이 아니면 토큰에 붙임  strncat(tokens[row], start, 1); // 현재 토큰에 1바이트를 붙임  start++;  }  }    else if(row>0 && is\_character(tokens[row - 1][strlen(tokens[row - 1]) - 1])){ // 앞 토큰의 마지막 인자가 문자일 경우  if(strstr(tokens[row - 1], "++") != NULL || strstr(tokens[row - 1], "--") != NULL) // 앞 토큰이 전위 연산자일 경우  return false;  memset(tmp, 0, sizeof(tmp));  strncpy(tmp, start, 2); // tmp에 start내용 2바이트 복사  strcat(tokens[row - 1], tmp); // tmp에 앞 토큰 내용 복사  start += 2; // start의 인덱스를 2바이트 이동  row--; //현재 토큰 개수 감소  }  else{ // 그 외의 경우  memset(tmp, 0, sizeof(tmp)); // 토큰에 2바이트만 저장  strncpy(tmp, start, 2);  strcat(tokens[row], tmp);  start += 2;  }  }  else if(!strncmp(start, "==", 2) || !strncmp(start, "!=", 2) || !strncmp(start, "<=", 2)  || !strncmp(start, ">=", 2) || !strncmp(start, "||", 2) || !strncmp(start, "&&", 2)  || !strncmp(start, "&=", 2) || !strncmp(start, "^=", 2) || !strncmp(start, "!=", 2)  || !strncmp(start, "|=", 2) || !strncmp(start, "+=", 2) || !strncmp(start, "-=", 2)  || !strncmp(start, "\*=", 2) || !strncmp(start, "/=", 2)){ // 비교 연산자일 경우  strncpy(tokens[row], start, 2); // 연산자 문자크기(2바이트) 만큼 저장  start += 2; // 2바이트 이동  }  else if(!strncmp(start, "->", 2)) // 구조체 포인터 변수 접근자일 경우  {  end = strpbrk(start + 2, op); // 다음 연산자가 나올때 까지 탐색  if(end == NULL) // 연산자가 나오지 않았을 경우  end = &str[strlen(str)]; // str의 마지막을 끝으로 지정  while(start < end){ // 다음 연산자까지 탐색  if(\*start != ' ') // 공백이 아니면  strncat(tokens[row - 1], start, 1); // 앞 토큰에 이어붙임  start++;  }  row--; // 현재 토큰 개수 감소  }  else if(\*end == '&') // 주소 포인터 접근자일 경우  {    if(row == 0 || (strpbrk(tokens[row - 1], op) != NULL)){ // 첫 토큰이거나 앞 토큰에 연산자가 없을 경우  end = strpbrk(start + 1, op); // 다음 연산자 탐색  if(end == NULL) // 다음 연산자가 없으면  end = &str[strlen(str)]; // str의 마지막을 end로 지정    strncat(tokens[row], start, 1); // 현재 토큰에 start의 1바이트('&')를 붙임  start++; // 다음 문자이동  while(start < end){ // 다음 연산자가 나올때 까지 탐색  if(\*(start - 1) == ' ' && tokens[row][strlen(tokens[row]) - 1] != '&') // start 1바이트 전이 공백이고 현재 토큰의 마지막 문자가 '&'일 경우  return false;  else if(\*start != ' ') // 공백이 아닐 경우  strncat(tokens[row], start, 1); // 1바이트를 현재 토큰에 이어 붙임  start++; // 다음 문자이동  }  }    else{  strncpy(tokens[row], start, 1); // 1바이트 이어붙이기  start += 1;  }    }  else if(\*end == '\*') // '\*' 이면  {  isPointer=0; // isPointer 초기화, '\*'이 무조건 포인터가 아닐 수도 있으므로  if(row > 0) // 토큰이 한개 이상이면  {    for(i = 0; i < DATATYPE\_SIZE; i++) { // 데이어타입 개수만큼  if(strstr(tokens[row - 1], datatype[i]) != NULL){ // 앞 토큰에 데이터타입이 있으면  strcat(tokens[row - 1], "\*"); // 앞 토큰에 '\*'삽입 (데이터 포인터)  start += 1; // 다음 문자 이동  isPointer = 1; // 포인터로 체크  break;  }  }  if(isPointer == 1) // 포인터일 경우  continue;  if(\*(start+1) !=0) // 다음 인자가 널문자가 아니면  end = start + 1; // end를 한칸 이동  // 더블 포인터?  if(row>1 && !strcmp(tokens[row - 2], "\*") && (all\_star(tokens[row - 1]) == 1)){ // 토큰이 2개 이상이고 2개 앞 토큰이 '\*'이며, 앞 토큰이 모두 '\*'이면  strncat(tokens[row - 1], start, end - start); // 앞 토큰에  row--; // 토큰 개수 감소  }      else if(is\_character(tokens[row - 1][strlen(tokens[row - 1]) - 1]) == 1){ // 앞 토큰의 마지막 인자가 문자면  strncat(tokens[row], start, end - start); // 현재 토큰에 다음 연산자 전까지 모두 입력  }    else if(strpbrk(tokens[row - 1], op) != NULL){ // 앞 토큰에 연산자가 있으면  strncat(tokens[row] , start, end - start); // 현재 토큰에 다음 연산자 전까지 모두 입력    }  else // 그외의 경우  strncat(tokens[row], start, end - start); // 현재 토큰에 다음 연산자 전까지 모두 입력  start += (end - start); // 다음 연산자까지 문자 이동  }  else if(row == 0) // 첫 토큰일 경우  {  if((end = strpbrk(start + 1, op)) == NULL){ // 다음 연산자가 없으면  strncat(tokens[row], start, 1); // 토큰에 1바이트('\*')을 붙이고  start += 1; // 다음 문자 이동  }  else{ // 연산자가 있으면  while(start < end){ // 다음 연산자까지  if(\*(start - 1) == ' ' && is\_character(tokens[row][strlen(tokens[row]) - 1])) // 이전 문자가 공백이고 현재 토큰의 마지막 인자가 문자일 경우  return false;  else if(\*start != ' ') // 공백일 경우  strncat(tokens[row], start, 1); // 현재 토큰에 내용 이어붙임  start++; // 다음 문자 이동  }  if(all\_star(tokens[row])) // 현재 토큰이 모두 '\*'문자로 이루어져 있을 경우 토큰 개수 감소  row--;    }  }  }  else if(\*end == '(') // 여는 괄호의 경우  {  lcount = 0;  rcount = 0;  if(row>0 && (strcmp(tokens[row - 1],"&") == 0 || strcmp(tokens[row - 1], "\*") == 0)){ // 토큰이 1개 이상이고, 앞 토큰이 '&' | '\*' 일 경우  while(\*(end + lcount + 1) == '(') // 괄호 내부 수식에 여는 괄호가 또 있을 경우  lcount++;  start += lcount; // 가장 안쪽 여는 괄호로 시작이동  end = strpbrk(start + 1, ")"); // 닫는 괄호가 나오기 전까지 탐색  if(end == NULL) // 닫는 괄호가 없는 경우  return false;  else{  while(\*(end + rcount +1) == ')') // 닫는 괄호 개수 체크  rcount++;  end += rcount;  if(lcount != rcount) // 괄호의 짝이 안맞을 경우  return false;  if( (row > 1 && !is\_character(tokens[row - 2][strlen(tokens[row - 2]) - 1])) || row == 1){ // 토큰이 2개 이상이고, 2개 앞 토큰의 마지막 인자가 문자가 아니거나 토큰이 1개일 경우  strncat(tokens[row - 1], start + 1, end - start - rcount - 1); // 앞 토큰에 닫는 괄호 전까지 모두 이어붙임  row--; // 토큰 개수 감소  start = end + 1; //가장 바깥쪽 닫는 괄호 다음으로 이동  }  else{ // 현재 토큰에 1바이트 이어붙임  strncat(tokens[row], start, 1);  start += 1;  }  }    }  else{ // 그 외엔 '(' 붙임  strncat(tokens[row], start, 1);  start += 1;  }  }  else if(\*end == '\"') // 따옴표면  {  end = strpbrk(start + 1, "\""); // 다음 따옴표의 위치를 탐색    if(end == NULL) // 다음 따옴표가 없을 경우  return false;  else{ // 현재 토큰에 따옴표 사이의 내용을 붙임  strncat(tokens[row], start, end - start + 1);  start = end + 1;  }  }  else{ // 그 외의 경우면    if(row > 0 && !strcmp(tokens[row - 1], "++")) // 토큰이 1개 이상이고 앞 토큰이 ++일 경우  return false;    if(row > 0 && !strcmp(tokens[row - 1], "--")) // 토큰이 1개 이상이고 앞 토큰이 --일 경우  return false;    strncat(tokens[row], start, 1); // 현재 토큰에 1바이트를 붙임  start += 1; // 문자 이동      if(!strcmp(tokens[row], "-") || !strcmp(tokens[row], "+") || !strcmp(tokens[row], "--") || !strcmp(tokens[row], "++")){ // 현재 토큰이 증감 연산자일 경우  if(row == 0) // 첫 토큰이면 row 감소  row--;    else if(!is\_character(tokens[row - 1][strlen(tokens[row - 1]) - 1])){ // 앞 토큰의 마지막 인자가 문자가 아닐 경우    if(strstr(tokens[row - 1], "++") == NULL && strstr(tokens[row - 1], "--") == NULL) // 앞 토큰에 전위 증감 연산자가 없으면 row 감소  row--;  }  }  }  }  else{ // start와 end가 다르면(단어의 경우)  if(all\_star(tokens[row - 1]) && row > 1 && !is\_character(tokens[row - 2][strlen(tokens[row - 2]) - 1])) //포인터인지 검사(row > 2)  row--;  if(all\_star(tokens[row - 1]) && row == 1) // 포인터인지 검사(row = 1)  row--;  for(i = 0; i < end - start; i++){ // 다음 토큰 문자가 오기 전까지 탐색  if(i > 0 && \*(start + i) == '.'){ // 구조체 접근이면  strncat(tokens[row], start + i, 1); // '.'을 바로 붙여줌  while( \*(start + i +1) == ' ' && i< end - start )  i++;  }  else if(start[i] == ' '){ // 공백일 경우 생략  while(start[i] == ' ')  i++;  break;  }  else // 그 외면 토큰에 붙임  strncat(tokens[row], start + i, 1);  }  if(start[0] == ' '){ // 공백일 경우 생략  start += i;  continue;  }  start += i; // 다음 토큰 시작 문자까지 인덱스 증가  }    strcpy(tokens[row], ltrim(rtrim(tokens[row]))); // 좌우 공백 제거 후 다시 저장  // 토큰이 1개 이상이고 현재 토큰의 마지막이 문자이고 앞 토큰이 데이터 타입이거나(변수 선언)  // 앞 토큰의 마지막 인자가 문자이거나 앞 토큰의 마지막 문자가 '.'이면  if(row > 0 && is\_character(tokens[row][strlen(tokens[row]) - 1])  && (is\_typeStatement(tokens[row - 1]) == 2  || is\_character(tokens[row - 1][strlen(tokens[row - 1]) - 1])  || tokens[row - 1][strlen(tokens[row - 1]) - 1] == '.' ) ){  if(row > 1 && strcmp(tokens[row - 2],"(") == 0) // 토큰이 2개 이상 있고 2개 앞 토큰이 여는 괄호일 때  {  if(strcmp(tokens[row - 1], "struct") != 0 && strcmp(tokens[row - 1],"unsigned") != 0) // 앞 토큰이 struct와 unsigned가 아닐 경우  return false;  }  else if(row == 1 && is\_character(tokens[row][strlen(tokens[row]) - 1])) { // 토큰이 한개이고, 현재 토큰의 마지막 인자가 문자일 경우  if(strcmp(tokens[0], "extern") != 0 && strcmp(tokens[0], "unsigned") != 0 && is\_typeStatement(tokens[0]) != 2) // 첫 토큰이 extern이 아니고, unsigned가 아니고, 데이터터타입이 아닐 경우  return false;  }  else if(row > 1 && is\_typeStatement(tokens[row - 1]) == 2){ // 토큰이 2개 이상 있고, 앞의 토큰이 데이터타입일 경우  if(strcmp(tokens[row - 2], "unsigned") != 0 && strcmp(tokens[row - 2], "extern") != 0)  return false;  }    }  if((row == 0 && !strcmp(tokens[row], "gcc")) ){ // 첫 토큰이 gcc일 경우  clear\_tokens(tokens); // 토큰을 비우고  strcpy(tokens[0], str); // 첫 토큰에 문장 전체를 넣어주고 종료  return 1;  }  row++;  } // 반복문 종료  if(all\_star(tokens[row - 1]) && row > 1 && !is\_character(tokens[row - 2][strlen(tokens[row - 2]) - 1])) // 앞 토큰이 별이고 2개 앞 토큰 마지막이 문자가 아닐 경우  row--;  if(all\_star(tokens[row - 1]) && row == 1) // 앞 토큰이 '\*'이면  row--;  for(i = 0; i < strlen(start); i++) // start에 남은 문자 처리  {  if(start[i] == ' ') // 공백일 경우  {  while(start[i] == ' ') // 공백이 아닐때까지 i 증가  i++;  if(start[0]==' ') { // 첫 인자가 공백이면  start += i; // start의 시작 위치 i까지 옮기고  i = 0; // i 초기화  }  else  row++; // 토큰 개수 증가    i--;  }  else  {  strncat(tokens[row], start + i, 1); // 현재 토큰에 1바이트를 붙임  if( start[i] == '.' && i<strlen(start)){ // 만약 '.'이고 위치가 start를 넘지 않았으면  while(start[i + 1] == ' ' && i < strlen(start)) // 다음 문자가 공백이 아닐때까지 i증가  i++;  }  }  strcpy(tokens[row], ltrim(rtrim(tokens[row]))); // 앞뒤 공백 제거 후 토큰에 추가  if(!strcmp(tokens[row], "lpthread") && row > 0 && !strcmp(tokens[row - 1], "-")){ // -lpthread 옵션이 있으면  strcat(tokens[row - 1], tokens[row]); // 두개를 붙여서 한 토큰에 넣고  memset(tokens[row], 0, sizeof(tokens[row]));  row--; // 토큰 개수 감소  }  else if(row > 0 && is\_character(tokens[row][strlen(tokens[row]) - 1])  && (is\_typeStatement(tokens[row - 1]) == 2  || is\_character(tokens[row - 1][strlen(tokens[row - 1]) - 1])  || tokens[row - 1][strlen(tokens[row - 1]) - 1] == '.') ){ // 토큰이 1개 이상이고 현재 토큰의 마지막 인자가 문자이고 데이터 타입이거나 앞 토큰의 마지막 인자가 문자이거나 '.'    if(row > 1 && strcmp(tokens[row-2],"(") == 0) // 토큰이 2개이상이고 2개 앞 토큰이 여는 괄호일 경우  {  if(strcmp(tokens[row-1], "struct") != 0 && strcmp(tokens[row-1], "unsigned") != 0) // 앞 토큰이 struct가 아니거나 unsigned가 아닐 경우  return false;  }  else if(row == 1 && is\_character(tokens[row][strlen(tokens[row]) - 1])) { // 토큰이 2개이고 현재 토큰의 마지막 인자가 문자일 경우  if(strcmp(tokens[0], "extern") != 0 && strcmp(tokens[0], "unsigned") != 0 && is\_typeStatement(tokens[0]) != 2) // 0번째 토큰이 extern이 아니거나 unsigned가 아니거나 데이터타입이 아닐 경우  return false;  }  else if(row > 1 && is\_typeStatement(tokens[row - 1]) == 2){ // 토큰이 2개 이상이고 앞 토큰이 데이터타입일 경우  if(strcmp(tokens[row - 2], "unsigned") != 0 && strcmp(tokens[row - 2], "extern") != 0) // 두개 앞 토큰이 unsigned가 아니고 extern도 아닐 경우  return false;  }  }  }  if(row > 0) // 토큰이 1개 이상일 경우  {    if(strcmp(tokens[0], "#include") == 0 || strcmp(tokens[0], "include") == 0 || strcmp(tokens[0], "struct") == 0){ // 첫번째 토큰이 #include이거나 include이거나 struct일 경우  clear\_tokens(tokens); // 토큰을 정리하고  strcpy(tokens[0], remove\_extraspace(str)); // 과도한 공백 제거  }  }  if(is\_typeStatement(tokens[0]) == 2 || strstr(tokens[0], "extern") != NULL){ // 첫 토큰이 데이터타입이거나 extern일 경ㅇ  for(i = 1; i < TOKEN\_CNT; i++){ // 토큰 개수만큼 순회  if(strcmp(tokens[i],"") == 0) // 토큰내에 정보가 없을 경우  break;  if(i != TOKEN\_CNT -1 ) // 토큰의 마지막이 아닐경우  strcat(tokens[0], " "); // 첫 토큰에 공백을 붙임  strcat(tokens[0], tokens[i]); // 첫 토큰에 i번째 토큰을 붙이고  memset(tokens[i], 0, sizeof(tokens[i])); // i본째 토큰 메모리 초기화  }  }      while((p\_str = find\_typeSpecifier(tokens)) != -1){ // 캐스팅 된 경우 체크  if(!reset\_tokens(p\_str, tokens))  return false;  }  while((p\_str = find\_typeSpecifier2(tokens)) != -1){ // struct 구조체 인 경우 체크  if(!reset\_tokens(p\_str, tokens))  return false;  }    return true;  } |

**2-5-4. node \*make\_tree(node \*root, char (\*tokens)[MINLEN], int \*idx, int parentheses)**

|  |
| --- |
| // char(\*tokens)[MINLEN] : 배열 포인터 -> 한줄에 35개의 인덱스를 갖는 배열을 가리킴  node \*make\_tree(node \*root, char (\*tokens)[MINLEN], int \*idx, int parentheses) // 주어진 노드의 파스트리 생성, 초기 idx:0, parentheses:괄호 짝 개수  {  node \*cur = root; // 노드 커서  node \*new;  node \*saved\_operator;  node \*operator;  int fstart;  int i;  while(1) // 토큰 탐색  {  if(strcmp(tokens[\*idx], "") == 0) // 1. 토큰 배열이 비어있을 경우  break;    if(!strcmp(tokens[\*idx], ")")) // 2. 토큰 배열에서 ')'를 만나면 괄호안의 문장을 마무리, 괄호내용을 포함하고 있는 부모 돌아감  return get\_root(cur);  else if(!strcmp(tokens[\*idx], ",")) // 3. 토큰 배열에서 ","를 만나면 부모로 돌아감  return get\_root(cur);  else if(!strcmp(tokens[\*idx], "(")) // 4. 토큰 배열에서 "("를 만날 경우  {    if(\*idx > 0 && !is\_operator(tokens[\*idx - 1]) && strcmp(tokens[\*idx - 1], ",") != 0){ // a. 앞의 문자가 연산자나 ","가 아닐 경우  fstart = true; // create^(fd, O\_RDONLY) 와 같은 케이스,  while(1)  {  \*idx += 1; // 다음 토큰 이동  if(!strcmp(tokens[\*idx], ")")) // 토큰이 ")"일 경우  break;    new = make\_tree(NULL, tokens, idx, parentheses + 1); // 자식 트리 생성    if(new != NULL){ // 자식 트리가 생성되었을 경우  if(fstart == true){  cur->child\_head = new; // 현재 노드의 자식 노드 연결  new->parent = cur; // 자식 노드의 부모를 현재 노드로 연결    fstart = false;  }  else{ // 함수 인자문이 아닐 때, 형제로 인식  cur->next = new; // 현재 노드의 다음 노드로 연결  new->prev = cur; // 다음 노드의 이전 노드를 현재 노드로 연결  }  cur = new; // 새로 만든 트리 노드로 이동  }  if(!strcmp(tokens[\*idx], ")")) // 토큰이 ")" 일 경우  break;  }  }  else{ // b. add(a,^(b+c)) | a +^(a+b) | (fd, O\_RDONLY) 와 같은 케이스, 기본 괄호 트리 생성 유형  \*idx += 1; // 다음 토큰 이동    new = make\_tree(NULL, tokens, idx, parentheses + 1); // 새로운 자식 트리 생성(재귀)  if(cur == NULL) // 현재 위치가 최상위 부모 노드일 경우  cur = new; // 자식 트리로 이동  else if(!strcmp(new->name, cur->name)){ // 자식 토큰과 현재의 토큰이 같다면 (++, --, &&, ||)  if(!strcmp(new->name, "|") || !strcmp(new->name, "||") // 자식의 토큰이 논리 연산자일 경우, (||, , (|, (&&, (&,  || !strcmp(new->name, "&") || !strcmp(new->name, "&&"))  {  cur = get\_last\_child(cur); // 자식중 가장 막내 노드로 이동  if(new->child\_head != NULL){ // 자식노드의 자식트리가 존재할 경우  new = new->child\_head; // 자식 노드도 최하위 자식의 마지막 형제로 이동  new->parent->child\_head = NULL; // 자식관계가 아닌 형제관계로 변경  new->parent = NULL; // 즉 비교 연산자 다음에 오는 토큰을 자식 관계로 생성되었지만 형제관계로 변경  new->prev = cur;  cur->next = new;  }  }  else if(!strcmp(new->name, "+") || !strcmp(new->name, "\*")) // 자식의 토큰이 수식 연산자 일 경우, + ( +, \* ( \*  {  i = 0; // 괄호 안 수식 내용 중 연산자 인덱스  while(1)  {  if(!strcmp(tokens[\*idx + i], "")) // 토큰이 비었을 경우  break;  if(is\_operator(tokens[\*idx + i]) && strcmp(tokens[\*idx + i], ")") != 0) // 해당 토큰이 연산자에 포함되고, ')'가 아닐 경우  break;  i++;  }    if(get\_precedence(tokens[\*idx + i]) < get\_precedence(new->name)) // 새로운 연산자와 기존 연산자 우선순위 비교, 새로운 연산자가 더 우선일경우  {  cur = get\_last\_child(cur); // 자식중 가장 막내 노드로 이동  cur->next = new; // 막내 노드 갱신(new)  new->prev = cur;  cur = new;  }  else // 기존 연산자가 더 우선일 경우  {  cur = get\_last\_child(cur); // 자식중 가장 막내 노드로 이동  if(new->child\_head != NULL){  new = new->child\_head;  new->parent->child\_head = NULL; // 새로운 연산자의 자식노드를 기존 연산자의 다음 형제로 이동  new->parent = NULL;  new->prev = cur;  cur->next = new;  }  }  }  else{  cur = get\_last\_child(cur); // 자식 중 가장 막내 노드로 이동  cur->next = new; // 막내 노드 갱신(new)  new->prev = cur;  cur = new;  }  }    else  {  cur = get\_last\_child(cur); // 자식중 가장 막내 노드로 이동  cur->next = new;  new->prev = cur;    cur = new;  }  }  }  else if(is\_operator(tokens[\*idx])) // 5. 토큰 배열에서 연산자를 만날 경우  {  if(!strcmp(tokens[\*idx], "||") || !strcmp(tokens[\*idx], "&&")  || !strcmp(tokens[\*idx], "|") || !strcmp(tokens[\*idx], "&")  || !strcmp(tokens[\*idx], "+") || !strcmp(tokens[\*idx], "\*"))  { // c. 논리 연산자의 경우  if(is\_operator(cur->name) == true && !strcmp(cur->name, tokens[\*idx])) // 현재 노드의 토큰이 연산자이고, 현재 노드의 토큰과 다음 토큰이 같은 연산자인 경우  operator = cur; // 연산자로 현재 노드를 지정    else // 아닐 경우  {  new = create\_node(tokens[\*idx], parentheses); // 새로운 노드 생성  operator = get\_most\_high\_precedence\_node(cur, new); // 기존 최우선 연산자와, 새로운 연산자의 우선순위 비교 후 연산자로 지정  if(operator->parent == NULL && operator->prev == NULL){ // 우선순위인 연산자 노드가 부모가 없고, 형제도 없으면  if(get\_precedence(operator->name) < get\_precedence(new->name)){ // 기존 연산자가 새로운 연산자보다 우선순위가 높을 경우  cur = insert\_node(operator, new); // 새로운 연산자를 기존 우선순위 연산자의 부모로 둠(트리의 잎에 가까울수록 우선순위 높음)  }  else if(get\_precedence(operator->name) > get\_precedence(new->name)) // 새로운 연산자가 기존연산자보다 우선순위가 높을 경우  {  if(operator->child\_head != NULL){ // 기존 연산자 노드의 자식이 존재하면 새로운 연산자 노드를 자식 연산자 노드들의 막내 자리에 두고, 막내노드가 새로운 연산자노드의 자식이 됨  operator = get\_last\_child(operator);  cur = insert\_node(operator, new);  }  }  else // 최우선 연산자랑 우선순위가 같거나 지정되어있지 않을 경우  {  operator = cur; // 최우선 연산자 지정    while(1)  {  if(is\_operator(operator->name) == true && !strcmp(operator->name, tokens[\*idx])) // 기존 연산자가 새로운 토큰이랑 같을 때  break;    if(operator->prev != NULL) // 기존 연산자의 형노드가 있을 경우  operator = operator->prev; // 기존 연산자의 형노드로 이동  else // 없을 경우  break;  }  if(strcmp(operator->name, tokens[\*idx]) != 0) // 기존 연산자와 새로운 토큰이 다를 경우  operator = operator->parent;  if(operator != NULL){ // 연산자 노드가 지정되어있을 경우  if(!strcmp(operator->name, tokens[\*idx])) // 기존 연산자와 새로운 토큰이 같을 경우  cur = operator; // 현재 노드는 연산자 노드로 지정  }  }  }  else  cur = insert\_node(operator, new);  }  }  else // d. 그 외에 연산자(<, >, /, -, <=, >=< ==, !=, ^, = ...)  {  new = create\_node(tokens[\*idx], parentheses); // 해당 토큰 노드 생성  if(cur == NULL) // 현재 가리키고 있는 노드가 없을 경우  cur = new; // 노드 생성 후 포인팅  else // 가리키고 있는 노드가 존재할 경우  {  operator = get\_most\_high\_precedence\_node(cur, new); // 현재 가리키는 노드와 연산자 우선순위 비교 후 선 순위 노드 할당, 우선순위(연산자 < 단어)  if(operator->parentheses > new->parentheses) // 기존의 연산자가 더 많은 괄호 안에 존재할 때 (new ((operator)))  cur = insert\_node(operator, new); // 새로운 연산자를 기존의 연산자 앞에 둠  else if(operator->parent == NULL && operator->prev == NULL){ // 우선 연산자가 최상위 노드일 경우(단어 일 경우 ?)    if(get\_precedence(operator->name) > get\_precedence(new->name)) // 기존연산자(단어) > 새 연산자  {  if(operator->child\_head != NULL){ // 단어의 자식이 존재할 경우    operator = get\_last\_child(operator); // 자식의 막내의 형노드로 삽입  cur = insert\_node(operator, new);  }  }    else  cur = insert\_node(operator, new); // 기존 연산자 < 새 연산자, 기존연산자의 부모로 삽입  }    else  cur = insert\_node(operator, new);  }  }  }  else // 6. 일반적인 단어일 경우  {  new = create\_node(tokens[\*idx], parentheses); // 노드 생성  if(cur == NULL) // 루트노드 시작이면 생성한 노드를 시작 노드로 지정  cur = new;  else if(cur->child\_head == NULL){ // 자식이 없다면  cur->child\_head = new; // 자식 노드로 지정  new->parent = cur;  cur = new;  }  else{ // 자식이 존재할 경우  cur = get\_last\_child(cur); // cur = 자식중의 막내노드  cur->next = new; // 막내 갱신  new->prev = cur;  cur = new;  }  }  \*idx += 1; // 인덱스 증가  }  return get\_root(cur); // 만들어진 트리의 루트노드를 반환  } |

**2-5-5. node \*change\_sibling(node \*parent)**

|  |
| --- |
| node \*change\_sibling(node \*parent) // 자식의 형제 노드들을 Swap (순서가 바뀐 동일 연산식 처리), 반환:parent  {  node \*tmp;    tmp = parent->child\_head;  // 자식 노드 포인팅  parent->child\_head = parent->child\_head->next;  parent->child\_head->parent = parent;  parent->child\_head->prev = NULL;  // prev, next 유동 변경  parent->child\_head->next = tmp;  parent->child\_head->next->prev = parent->child\_head;  parent->child\_head->next->next = NULL;  parent->child\_head->next->parent = NULL;  return parent;  } |

**2-5-6. node \*create\_node(char \*name, int parentheses)**

|  |
| --- |
| node \*create\_node(char \*name, int parentheses) // 토큰 내용, 괄호개수 정보를 가진 노드 생성, 반환:new node  {  node \*new;  new = (node \*)malloc(sizeof(node));  new->name = (char \*)malloc(sizeof(char) \* (strlen(name) + 1));  strcpy(new->name, name);  new->parentheses = parentheses;  new->parent = NULL;  new->child\_head = NULL;  new->prev = NULL;  new->next = NULL;  return new;  } |

**2-5-7. int get\_precedence(char \*op)**

|  |
| --- |
| int get\_precedence(char \*op) // 연산자의 우선순위 반환, 반환:우선순위  {  int i;  for(i = 2; i < OPERATOR\_CNT; i++){  if(!strcmp(operators[i].operator, op))  return operators[i].precedence;  }  return false;  } |

**2-5-8. int is\_operator(char \*op)**

|  |
| --- |
| int is\_operator(char \*op) // 연산자 체크, 연산자:true, 그외:false  {  int i;  for(i = 0; i < OPERATOR\_CNT; i++)  {  if(operators[i].operator == NULL)  break;  if(!strcmp(operators[i].operator, op)){  return true;  }  }  return false;  } |

**2-5-9. void print(node \*cur)**

|  |
| --- |
| void print(node \*cur) // 트리 출력  {  if(cur->child\_head != NULL){ // 자식이 존재할 경우 자식을 출력하고 개행  print(cur->child\_head);  printf("\n");  }  if(cur->next != NULL){ // 형제의 경우 형제를 출력하고 띄어쓰기  print(cur->next);  printf("\t");  }  printf("%s", cur->name); // 토큰 출력  } |

**2-5-10. node \*get\_operator(node \*cur)**

|  |
| --- |
| node \*get\_operator(node \*cur) // 형제들의 부모 노드를 반환, 반환:parent  {  if(cur == NULL)  return cur;  if(cur->prev != NULL)  while(cur->prev != NULL)  cur = cur->prev;  return cur->parent;  } |

**2-5-11. char \*get\_root(node \*cur)**

|  |
| --- |
| node \*get\_root(node \*cur) // 트리의 최상위 노드를 반환, 반환:root  {  if(cur == NULL)  return cur;  while(cur->prev != NULL)  cur = cur->prev;  if(cur->parent != NULL)  cur = get\_root(cur->parent);  return cur;  } |

**2-5-12. node \*get\_high\_precedence\_node(node \*cur, node \*new)**

|  |
| --- |
| node \*get\_high\_precedence\_node(node \*cur, node \*new) // 두 노드 중 우선순위가 가장 높은 연산자 노드를 반환(기존 연산자만 리턴)  {  if(is\_operator(cur->name)) // 일단 비교 후, 기존 연산자 노드가 우선순위가 높으면  if(get\_precedence(cur->name) < get\_precedence(new->name))  return cur; // 기존 연산자 노드 리턴  if(cur->prev != NULL){ // 그런데 새로운 연산자 노드가 우선순위가 높을 경우, 기존 연산자 노드의 형제들 탐색  while(cur->prev != NULL){ // 형제중 맏형 노드로 이동 후  cur = cur->prev;    return get\_high\_precedence\_node(cur, new); // 맏형노드와 우선순위 비교  }  if(cur->parent != NULL) // 맏형 노드와도 비교 후 새 연산자 노드가 더 클 경우, 부모 노드 호출  return get\_high\_precedence\_node(cur->parent, new); // 부모 노드와 우선순위 비교  }  if(cur->parent == NULL) // 더이상 부모 노드가 없을경우  return cur; // 현재 노드 반환  } |

**2-5-13. node \*get\_most\_high\_precedence\_node(node \*cur, node \*new)**

|  |
| --- |
| node \*get\_most\_high\_precedence\_node(node \*cur, node \*new) // 새 연산자와 기존 연산자 노드,형제,부모 비교, 새연산자 노드가 우선순위가 가장 높을경우 새 연산자 노드도 리턴  {  node \*operator = get\_high\_precedence\_node(cur, new); // 기존연산자(형제, 부모 포함)와 새 연산자 우선순위 비교  node \*saved\_operator = operator;  while(1) // 가장 우선순위가 높은 연산자가 판별되고  {  if(saved\_operator->parent == NULL) // 가장 우선순위의 노드의 부모가 존재하지 않으면 (새 연산자 혹은 기존 최우선 연산자), 더이상 비교 불가  break;  if(saved\_operator->prev != NULL) // 가장 우선순위 노드의 형제들이 존재하면  operator = get\_high\_precedence\_node(saved\_operator->prev, new); // 형제노드들과 우선순위 비교  else if(saved\_operator->parent != NULL) // 가장 우선순위 노드의 부모 노드가 존재할 경우  operator = get\_high\_precedence\_node(saved\_operator->parent, new); // 부모노드와 우선순위 비교  saved\_operator = operator; // 형제 노드도 없고, 부모노드도 없을 경우  }    return saved\_operator; // 결정된 최우선 노드 반환  } |

**2-5-14. node \*insert\_node(node \*old, node \*new)**

|  |
| --- |
| node \*insert\_node(node \*old, node \*new) // 노드를 삽입, 삽입한 위치이후 노드는 자식관계로 전환, 삽입한 노드 반환  {  if(old->prev != NULL){ // 형제가 존재할 경우 삽입할 노드와 형제관계로 이어줌  new->prev = old->prev; // 기존 노드의 연결을 새노드가 물려받음  old->prev->next = new;  old->prev = NULL;  }  new->child\_head = old; // 새 노드의 자식을 기존 노드로 변경  old->parent = new; // 기존 노드의 부모를 새 노드로 변경  return new;  } |

**2-5-15. node \*get\_last\_child(node \*cur)**

|  |
| --- |
| node \*get\_last\_child(node \*cur) // 인자로 주어진 노드의 자식 중 막내를 리턴  {  if(cur->child\_head != NULL)  cur = cur->child\_head;  while(cur->next != NULL)  cur = cur->next;  return cur;  } |

**2-5-16. int get\_sibling\_cnt(node \*cur)**

|  |
| --- |
| int get\_sibling\_cnt(node \*cur) // 인자로 주어진 노드의 형제 개수 반환, 반환:형제 수(자기자신 미포함)  {  int i = 0;  while(cur->prev != NULL)  cur = cur->prev;  while(cur->next != NULL){  cur = cur->next;  i++;  }  return i;  } |

**2-5-17. void free\_node(node \*cur)**

|  |
| --- |
| void free\_node(node \*cur) // 노드 연결 해제  {  if(cur->child\_head != NULL)  free\_node(cur->child\_head);  if(cur->next != NULL)  free\_node(cur->next);  if(cur != NULL){  cur->prev = NULL;  cur->next = NULL;  cur->parent = NULL;  cur->child\_head = NULL;  free(cur);  }  } |

**2-5-18. int is\_character(char c)**

|  |
| --- |
| int is\_character(char c) // 문자인지 체크하는 함수, 문자(숫자, 알파벳):1, 그외:0  {  return (c >= '0' && c <= '9') || (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');  } |

**2-5-19. int is\_typeStatement(char \*str)**

|  |
| --- |
| int is\_typeStatement(char \*str) // 타입정의문 유무 확인  {  char \*start;  char str2[BUFLEN] = {0};  char tmp[BUFLEN] = {0};  char tmp2[BUFLEN] = {0};  int i;    start = str;  strncpy(str2,str,strlen(str));  remove\_space(str2);  while(start[0] == ' ')  start += 1;  /\*  if(strstr(str2, "gcc") != NULL)  {  strncpy(tmp2, start, strlen("gcc"));  if(strcmp(tmp2,"gcc") != 0)  return 0;  else  return 2;  }  \*/  for(i = 0; i < DATATYPE\_SIZE; i++) // 데이터 타입 리스트중  {  if(strstr(str2,datatype[i]) != NULL) // 해당하는 데이터 타입이 문자열에 존재할 경우  {  strncpy(tmp, str2, strlen(datatype[i])); // 데이터타입을 임시 배열에 저장  strncpy(tmp2, start, strlen(datatype[i]));    if(strcmp(tmp, datatype[i]) == 0) // str이 특정 자료형으로 시작한다면  if(strcmp(tmp, tmp2) != 0)  return 0; // 단순 자료형이 포함된 문자열 일 경우  else  return 2; // 변수 선언일 경우(자료형으로 시작하는 문장)  }  }  return 1; // 아무것도 아닐 경우(캐스팅의 경우)  } |

**2-5-20. int find\_typeSpecifier(char tokens[TOKEN\_CNT][MINLEN])**

|  |
| --- |
| int find\_typeSpecifier(char tokens[TOKEN\_CNT][MINLEN]) // 형변환을 확인하고 그에 해당하는 토큰 넘버 반환, 오류: -1  {  int i, j;  for(i = 0; i < TOKEN\_CNT; i++) // 데이터 타입 탐색  {  for(j = 0; j < DATATYPE\_SIZE; j++)  {  if(strstr(tokens[i], datatype[j]) != NULL && i > 0)  {  if(!strcmp(tokens[i - 1], "(") && !strcmp(tokens[i + 1], ")")  && (tokens[i + 2][0] == '&' || tokens[i + 2][0] == '\*'  || tokens[i + 2][0] == ')' || tokens[i + 2][0] == '('  || tokens[i + 2][0] == '-' || tokens[i + 2][0] == '+'  || is\_character(tokens[i + 2][0]))) // 캐스팅처럼 (데이터타입) 이후 문자 혹은 연산자가 오면 해당 토큰 넘버 반환  return i;  }  }  }  return -1;  } |

**2-5-21. int all\_star(char \*str)**

|  |
| --- |
| int all\_star(char \*str) // str이 모두 '\*'일 경우:1 아닐경우: 0  {  int i;  int length= strlen(str);    if(length == 0)  return 0;    for(i = 0; i < length; i++)  if(str[i] != '\*')  return 0;  return 1;  } |

**2-5-22. int all\_character(char \*str)**

|  |
| --- |
| int all\_character(char \*str) // 받아온 문자열 중에 알파벳 혹은 숫자가 들어있으면 1, 아니면 0  {  int i;  for(i = 0; i < strlen(str); i++)  if(is\_character(str[i]))  return 1;  return 0;    } |

**2-5-23. int reset\_tokens(int start, char tokens[TOKEN\_CNT][MINLEN])**

|  |
| --- |
| int reset\_tokens(int start, char tokens[TOKEN\_CNT][MINLEN]) // 형변환, struct와 같은 토큰을 읽었을 경우 문법 확인 후 결과 리턴, start = 형변환 | struct가 위치한 토큰 넘버  {  int i;  int j = start - 1; // 커서역할  int lcount = 0, rcount = 0;  int sub\_lcount = 0, sub\_rcount = 0;  if(start > -1){  if(!strcmp(tokens[start], "struct")) { // 불러온 토큰이 struct 라면  strcat(tokens[start], " ");  strcat(tokens[start], tokens[start+1]); // struct + ' ' + 변수명과 같은 형태로 문자열 합침  for(i = start + 1; i < TOKEN\_CNT - 1; i++){ // 토큰 배열 재정리  strcpy(tokens[i], tokens[i + 1]);  memset(tokens[i + 1], 0, sizeof(tokens[0]));  }  }  else if(!strcmp(tokens[start], "unsigned") && strcmp(tokens[start+1], ")") != 0) { // 현재 토큰이 unsigned이고 다음 토큰이 ')'일 경우  strcat(tokens[start], " "); // unsigned + ' ' + 그 다음 토큰 두개를 하나의 토큰으로 합침(ex. unsigned char i)  strcat(tokens[start], tokens[start + 1]);  strcat(tokens[start], tokens[start + 2]);  for(i = start + 1; i < TOKEN\_CNT - 1; i++){ // 토큰 배열 재정리  strcpy(tokens[i], tokens[i + 1]);  memset(tokens[i + 1], 0, sizeof(tokens[0]));  }  }  j = start + 1;  while(!strcmp(tokens[j], ")")){ // 현재 토큰 뒤에 ')'가 나온 횟수 측정  rcount ++;  if(j==TOKEN\_CNT)  break;  j++;  }    j = start - 1;  while(!strcmp(tokens[j], "(")){ // 현재 토큰 앞에 '(' 나온 횟수 측정  lcount ++;  if(j == 0)  break;  j--;  }  if( (j!=0 && is\_character(tokens[j][strlen(tokens[j])-1]) ) || j==0) // '('의 왼쪽 끝의 왼쪽에 char형 변수가 있거나 아무것도 없는 경우  lcount = rcount; // 괄호의 짝이 같다고 판단  if(lcount != rcount) // 괄호의 짝이 다를경우  return false;  if( (start - lcount) >0 && !strcmp(tokens[start - lcount - 1], "sizeof")){ // 토큰을 감싸고 있는 왼쪽괄호 전에 sizeof가 있을 경우  return true;  }    else if((!strcmp(tokens[start], "unsigned") || !strcmp(tokens[start], "struct")) && strcmp(tokens[start+1], ")")) { // unsigned나 struct의 바로 뒤에 닫는 괄호가 나온 경우  strcat(tokens[start - lcount], tokens[start]); // 불필요하게 씌어진 괄호들을 제거  strcat(tokens[start - lcount], tokens[start + 1]);  strcpy(tokens[start - lcount + 1], tokens[start + rcount]);    for(int i = start - lcount + 1; i < TOKEN\_CNT - lcount -rcount; i++) { // 제거한 이후의 토큰들을 정렬  strcpy(tokens[i], tokens[i + lcount + rcount]);  memset(tokens[i + lcount + rcount], 0, sizeof(tokens[0])); // 그 이후의 토큰 초기화  }  }  else{ // 그 외의 경우  if(tokens[start + 2][0] == '('){ // // 현재 토큰으로부터 2번째 뒤의 토큰에 여는 괄호가 있을 경우  j = start + 2;  while(!strcmp(tokens[j], "(")){ // 그 뒤의 여는괄호가 닫힐 때까지  sub\_lcount++;  j++;  }  if(!strcmp(tokens[j + 1],")")){ // 여는 괄호 바로 다음 토큰이 닫는괄호 일경우  j = j + 1;  while(!strcmp(tokens[j], ")")){  sub\_rcount++;  j++;  }  }  else  return false;  if(sub\_lcount != sub\_rcount) // 내부 괄호의 개수가 다른 경우  return false;    strcpy(tokens[start + 2], tokens[start + 2 + sub\_lcount]); // start + 2 이후에 들어간 여는 괄호를 제거  for(int i = start + 3; i<TOKEN\_CNT; i++)  memset(tokens[i], 0, sizeof(tokens[0])); // 그 이후의 토큰은 의미 없으므로 초기화  }  strcat(tokens[start - lcount], tokens[start]); // 괄호 제거  strcat(tokens[start - lcount], tokens[start + 1]);  strcat(tokens[start - lcount], tokens[start + rcount + 1]);    for(int i = start - lcount + 1; i < TOKEN\_CNT - lcount -rcount -1; i++) { // 토큰 재정리  strcpy(tokens[i], tokens[i + lcount + rcount +1]);  memset(tokens[i + lcount + rcount + 1], 0, sizeof(tokens[0])); // 이후 토큰 초기화  }  }  }  return true;  } |

**2-5-24. void clear\_tokens(char tokens[TOKEN\_CNT][MINLEN])**

|  |
| --- |
| void clear\_tokens(char tokens[TOKEN\_CNT][MINLEN]) // 토큰 배열 초기화  {  int i;  for(i = 0; i < TOKEN\_CNT; i++)  memset(tokens[i], 0, sizeof(tokens[i]));  } |

**2-5-25. char \*rtrim(char \*\_str)**

|  |
| --- |
| char \*rtrim(char \*\_str) // 문자열 오른쪽 공백 제거  {  char tmp[BUFLEN];  char \*end;  strcpy(tmp, \_str);  end = tmp + strlen(tmp) - 1;  while(end != \_str && isspace(\*end))  --end;  \*(end + 1) = '\0';  \_str = tmp;  return \_str;  } |

**2-5-26. char \*ltrim(char \*\_str)**

|  |
| --- |
| char \*ltrim(char \*\_str) // 문자열 왼쪽 공백 제거  {  char \*start = \_str;  while(\*start != '\0' && isspace(\*start))  ++start;  \_str = start;  return \_str;  } |

**2-5-27. char \*remove\_extraspace(char \*str)**

|  |
| --- |
| char\* remove\_extraspace(char \*str) // 잔여 공백 제거  {  int i;  char \*str2 = (char\*)malloc(sizeof(char) \* BUFLEN);  char \*start, \*end;  char temp[BUFLEN] = "";  int position;  if(strstr(str,"include<")!=NULL){ // 답안 내용에 include<가 있는지 확인  start = str; // 시작위치  end = strpbrk(str, "<"); // <부터 위치 저장  position = end - start;    strncat(temp, str, position); // #include  strncat(temp, " ", 1); // #include + ' '  strncat(temp, str + position, strlen(str) - position + 1); // #include + ' ' + <filename>  str = temp;  }    for(i = 0; i < strlen(str); i++)  {  if(str[i] ==' ') // 문자가 공백일 경우  {  if(i == 0 && str[0] ==' ') // 문자열 전의 공백일 경우  while(str[i + 1] == ' ') // 인덱스 증가  i++;  else{  if(i > 0 && str[i - 1] != ' ') // 문자열 내부의 공백일 경우  str2[strlen(str2)] = str[i];  while(str[i + 1] == ' ') // 연속적 공백일 경우  i++;  }  }  else // 공백이 없을 경우  str2[strlen(str2)] = str[i];  }  return str2; // 불필요한 공백이 제거된 문자열 리턴  } |

**2-5-28. void remove\_space(char \*str)**

|  |
| --- |
| void remove\_space(char \*str) // 문자열 공백 제거  {  char\* i = str;  char\* j = str;    while(\*j != 0)  {  \*i = \*j++;  if(\*i != ' ')  i++;  }  \*i = 0;  } |

**2-5-29. int check\_brackets(char \*str)**

|  |
| --- |
| int check\_brackets(char \*str) // 괄호 짝 확인 함수, 맞음:1, 틀림:0  {  char \*start = str;  int lcount = 0, rcount = 0;    while(1){  if((start = strpbrk(start, "()")) != NULL){  if(\*(start) == '(')  lcount++;  else  rcount++;  start += 1;  }  else  break;  }  if(lcount != rcount)  return 0;  else  return 1;  } |

**2-5-30. int get\_token\_cnt(char tokens[TOKEN\_CNT][MINLEN])**

|  |
| --- |
| int get\_token\_cnt(char tokens[TOKEN\_CNT][MINLEN]) // 토큰의 개수 반환  {  int i;    for(i = 0; i < TOKEN\_CNT; i++)  if(!strcmp(tokens[i], ""))  break;  return i;  } |

**2-6. Makefile**

|  |
| --- |
| # 실행 파일 생성  ssu\_score : main.o ssu\_score.o blank.o  gcc main.o ssu\_score.o blank.o -o ssu\_score  # 오브젝트 파일 생성  main.o : main.c ssu\_score.h  gcc -c main.c  ssu\_score.o : ssu\_score.c ssu\_score.h blank.h  gcc -c ssu\_score.c  blank.o : blank.c blank.h  gcc -c blank.c  # 실행 후 생성된 파일 전부 삭제  clean :  rm \*.o # 오브젝트 파일 삭제  rm ssu\_score # 실행 파일 삭제  rm $(PWD)/\*.csv # 점수 테이블 삭제  rm $(PWD)/ANS\_DIR/\*.exe # 정답 실행 파일 삭제  rm $(PWD)/ANS\_DIR/\*.stdout # 정답 실행 결과 삭제  rm $(PWD)/\*.txt # 생성된 텍스트 파일 삭제  rm $(PWD)/STD\_DIR/\*/\*.stdout # 학생 실행 결과 삭제  rm $(PWD)/STD\_DIR/\*/\*.stdexe # 학생 실행 파일 삭제 |

1. **설계**
2. **구현**

**4-1. 수정**

**4-1-1. 전역변수**

기존에 ssu\_score( )함수 내부에서 존재하던 지역변수 saved\_path[]를 전역변수로 선언하였고, 옵션 변수들 중 pOption과 cOption을 iOptiond, mOption으로 변경하였음.

**4-1-2. void ssu\_score(int argc, char \*argv[])**

초반 -c 옵션을 체크하는 조건문을 -i옵션을 체크하는 조건문으로 변경하였음. 기존의 채점 테이블 파일이 존재할 경우 바로 틀린 문제 목록을 출력함. 그리고 마지막으로 다시 -i옵션을 체크하는 지점을 생성하여, 초반에 테이블 생성 전일 경우, 옵션을 실행하지 않고, 채점 완료 후 테이블이 생성된 후 옵션을 실행함. 또한 후반부에 채점 직전 -m 옵션을 체크하는 조건문을 반영하여 점수를 변경하는 지점을 새로 추가하였음.

**4-1-3. int check\_option(int argc, char \*argv[])**

getopt( )함수가 탐색하는 기존의 -c, -p 옵션을 제거하고, -m, -i옵션을 추가하였음.

**4-1-4. void set\_scoreTable(char \*ansDir)**

score\_table.csv파일의 위치를 $(PWD)/ANS\_DIR/score\_table.csv에서 $(PWD)/score\_table.csv로 변경

**4-1-5. viud make\_scoreTable(char \*ansDir)**

변경된 정답 디렉토리 하위에 존재하는 문제들의 위치에 맞게 정답 디렉토리 파싱 방법 변경

**4-1-6. void score\_student() … 2-3-17**

기존에 존재하던 -p 옵션 확인 조건문만 제거

**4-1-7. void score\_student(int fd, char \*id) … 2-3-18**

-p옵션 확인 조건문을 제거하고, 최종 출력문을 -p 옵션이 켜져 있을 경우 출력문으로 고정

**4-1-8. int score\_blank(char \*id, char \*filename)**

정답 빈칸 문제 채점을 위한 경로 변수 tmp를 ansDir/qname/filename에서 ansDir/filename으로 문자열 변경

**4-1-9. double compile\_program(char \*id, char \*filename)**

tmp\_f, tmp\_e의 경로를 ansDir/qname/filename, ansDir/qname/qname에서 ansDir/filename, ansDir/qname으로 변경

**4-1-10. int execute\_program(char \*id, char \*filename)**

정답 프로그램 실행을 위한 ans\_fname, tmp의 문자열을 ansDir/qname/qname에서 ansDir/qname으로 변경

**4-1-11. Makefile**

기존 Makefile에서 변경점은 make clean 명령어 시 기존에는 오브젝트 파일들과 실행 파일만 제거하였지만, 답안 프로그램 실행 후 생성되는 텍스트 파일들과 테이블 파일, 실행파일, 실행결과 파일들을 제거한다.

**4-2. 구현**

**4-2-1. void do\_iOption(char (\*ids)[FILELEN])**

학생 별 틀린 문제에 대한 목록을 출력해주는 옵션으로, score.csv파일을 참조한다. 만약 score.csv파일이 존재하지 않을 경우, iOption변수를 그대로 놔두고 리턴을 한다. 이후 학생 답안의 채점이 모두 완료되고 score.csv파일이 생성 되면그 때 다시 해당 변수를 확인 한 뒤 1일 경우 다시 옵션을 실행한다. 만약 학생, 정답 디렉토리를 인자로 주지 않고 옵션만 실행 할 경우, score.csv파일이 존재 할 경우에는 인자로 주어진 학번이 틀린 문제 목록을 리스트로 출력하고, 해당 옵션 변수를 0으로 바꾼 뒤 반환한다. 반대로 파일이 존재하지 않을 경우

**4-2-2. int get\_header\_idx(char \*header, int comma\_cnt)**

이 함수는 score.csv파일에서 첫번째 행을 콤마(,)를 기준으로 문자열 인덱스를 결정하며, 인자로 주어지는 첫번째 행 문자열 header와 쉼표 개수 comma\_cnt를 통해 해당하는 문자열 시작 인덱스를 반환한다. 쉼표 개수 만큼 카운트 하지 못했거나, 쉼표가 존재하지 않을 경우 문자열의 끝을 반환한다.

**4-2-3. char \*get\_header\_char(char \*header, int idx)**

이 함수는 score.csv파일에서 첫번째 행을 콤마(,)를 기준으로 문자열 인덱스를 결정하며, 인자로 주어지는 idx를 기준으로 해당하는 문제 파일명 문자열을 반환한다.

**4-2-4. void do\_mOption()**

기존에 등록한 문제들에 대한 점수 수정기능을 제공해주는 옵션으로, score\_table.csv파일을 참조한다. 해당 옵션은 score\_table.csv파일이 생성되고 난 뒤 옵션 확인을 하며, 해당하는 문제 번호의 점수를 수정 할 경우, 기존의 점수를 출력해주고 새로운 점수를 입력 받은 뒤, score\_table.csv파일을 새로 작성한다. 문제 수정은 개수 제한 없이 이루어 질 수 있으며, “no”를 입력하면 종료한다.

**4-2-5. int rescore(int line, double new\_score)**

이 함수는 score\_table.csv파일을 참조하여 인자로 주어진 line 변수를 통해 해당하는 라인 전까지 score\_table.csv파일을 복사하고, 해당되는 라인에서는 새로운 점수 값을 가지고 있는 new\_score을 쓴다. 이후 다시 파일의 끝까지 내용을 복사한다.

**4-2-6. void file\_remove(const char \*file, \_Bool isDir)**

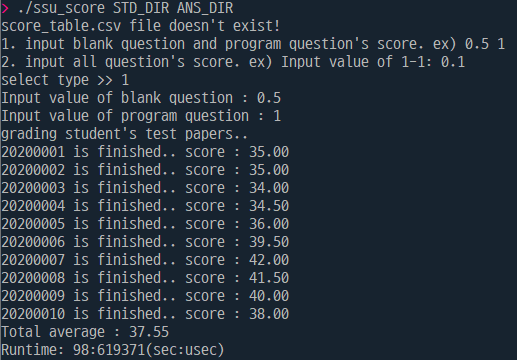
인자로 주어진 file변수로 rm file\_path명령어 문자열을 만들어 실행하며, \_isDir 변수를 통해 디렉토리일 경우 -r옵션을 준다.

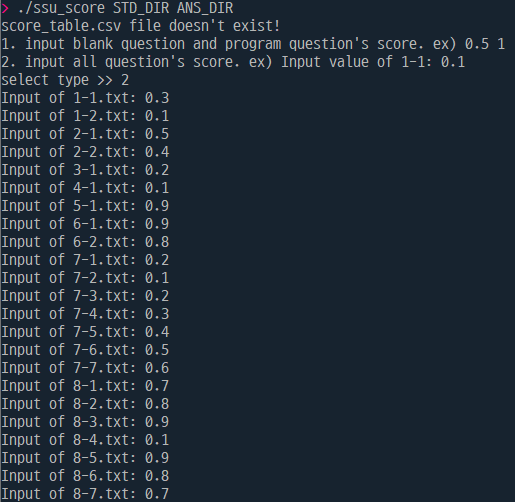
**4-2-7. void file\_copy(const char \*from, const char \*to)**

인자로 주어진 from과 to변수로 cp from to 명령어 문자열을 만들어 실행한다.

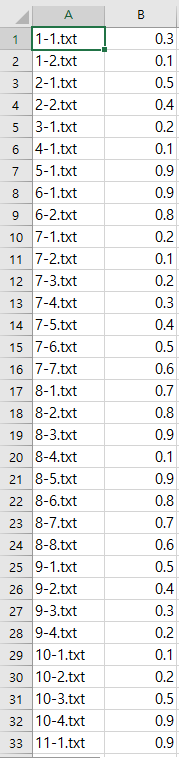
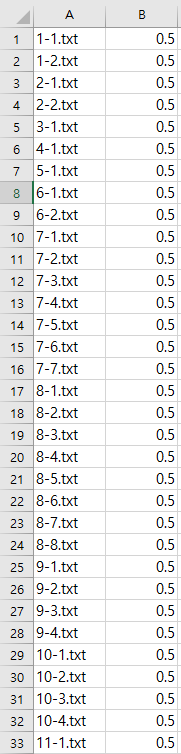
1. **테스트 및 결과**

**5-1. 점수 테이블 파일 생성 (상:1번, 하:2번)**

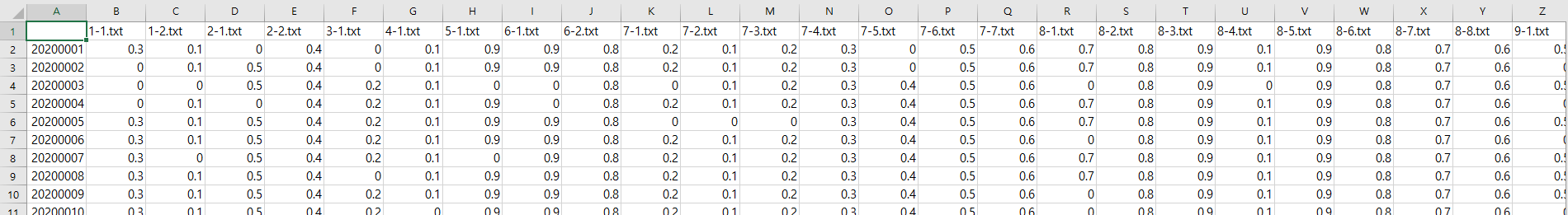




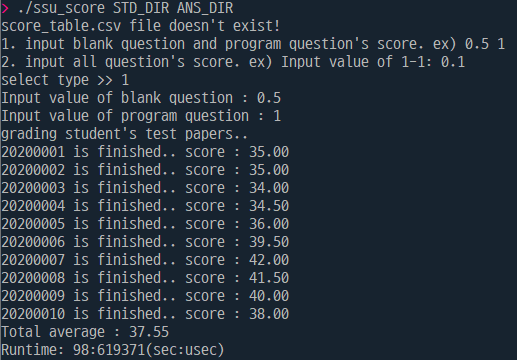
**5-2. 점수 테이블 파일 결과 (좌:1번, 우:2번)**



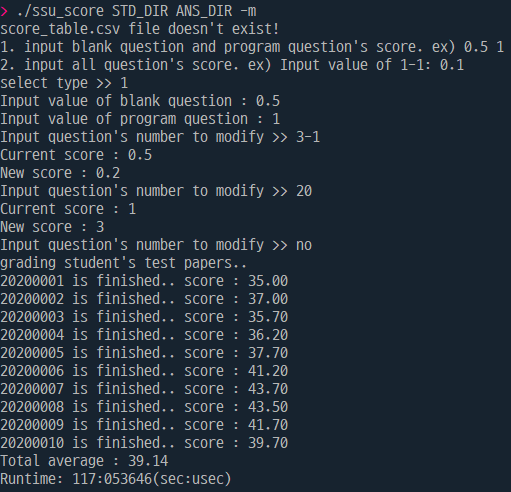
**5-3. 채점 결과 파일**



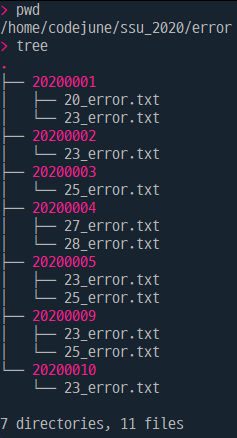
**5-4. 옵션 없이 실행하는 경우**

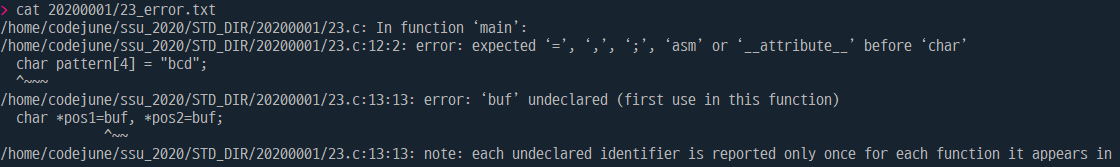


**5-5. –m 옵션 실행**

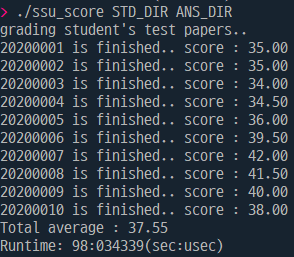


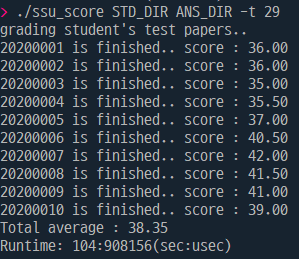
**5-6. –e 옵션 실행**



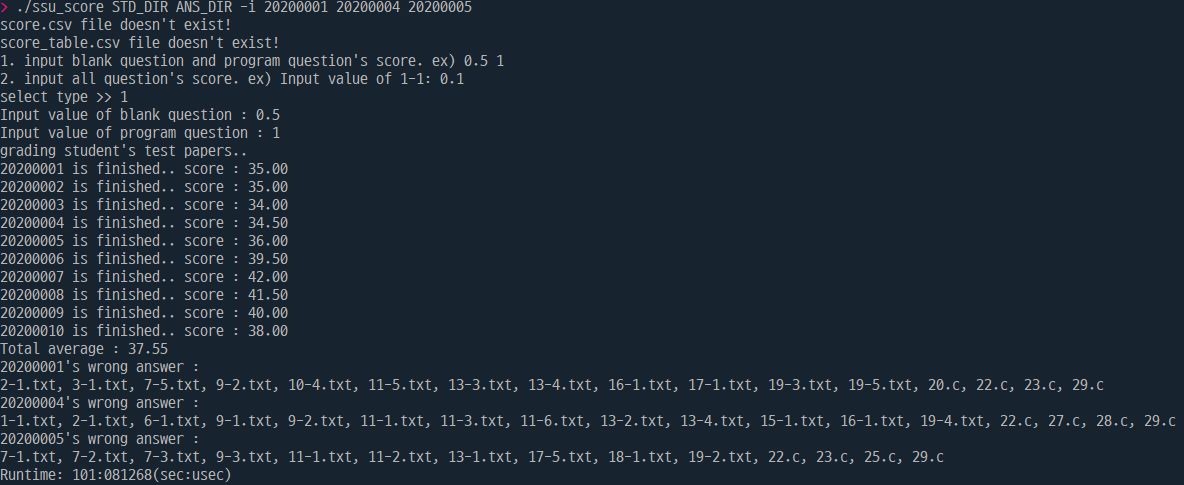


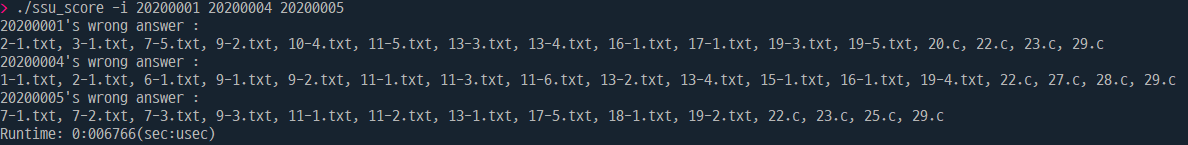
**5-7. –t 옵션 실행 (상:실행전, 하:실행후)**



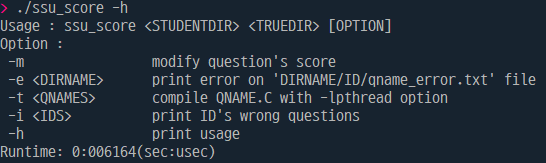


**5-8. –i 옵션 실행 (상:실행전, 하:실행후)**

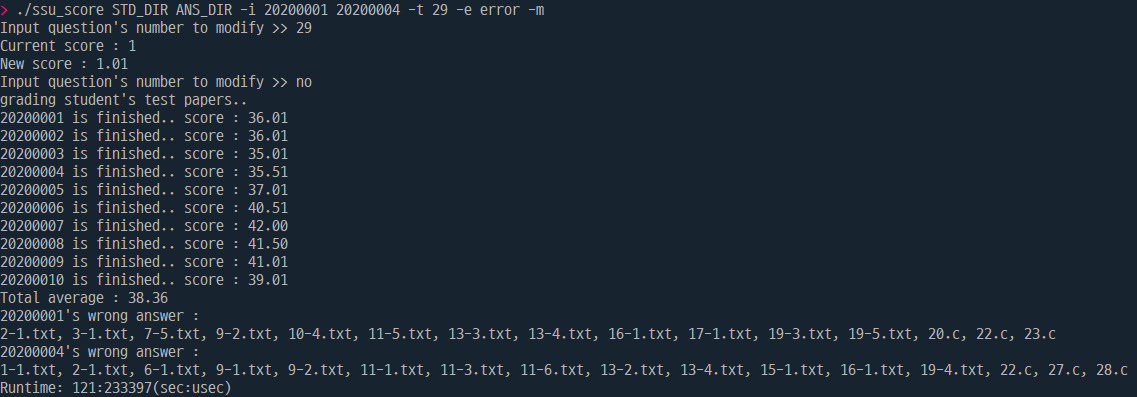




**5-9. –h 옵션 실행**



**5-9. 여러 옵션 동시 실행**



**5-9. 가변 인자 최대 개수를 초과한 경우 실행**

1. **소스코드**