프로그래밍 언어
구현 과제 #1
(Recursive-Descent Parser)

학번	20201466
이름	박사성

※ JAVA 파일 이클립스 실행 시 패키지 이름은 RD로 하였습니다.

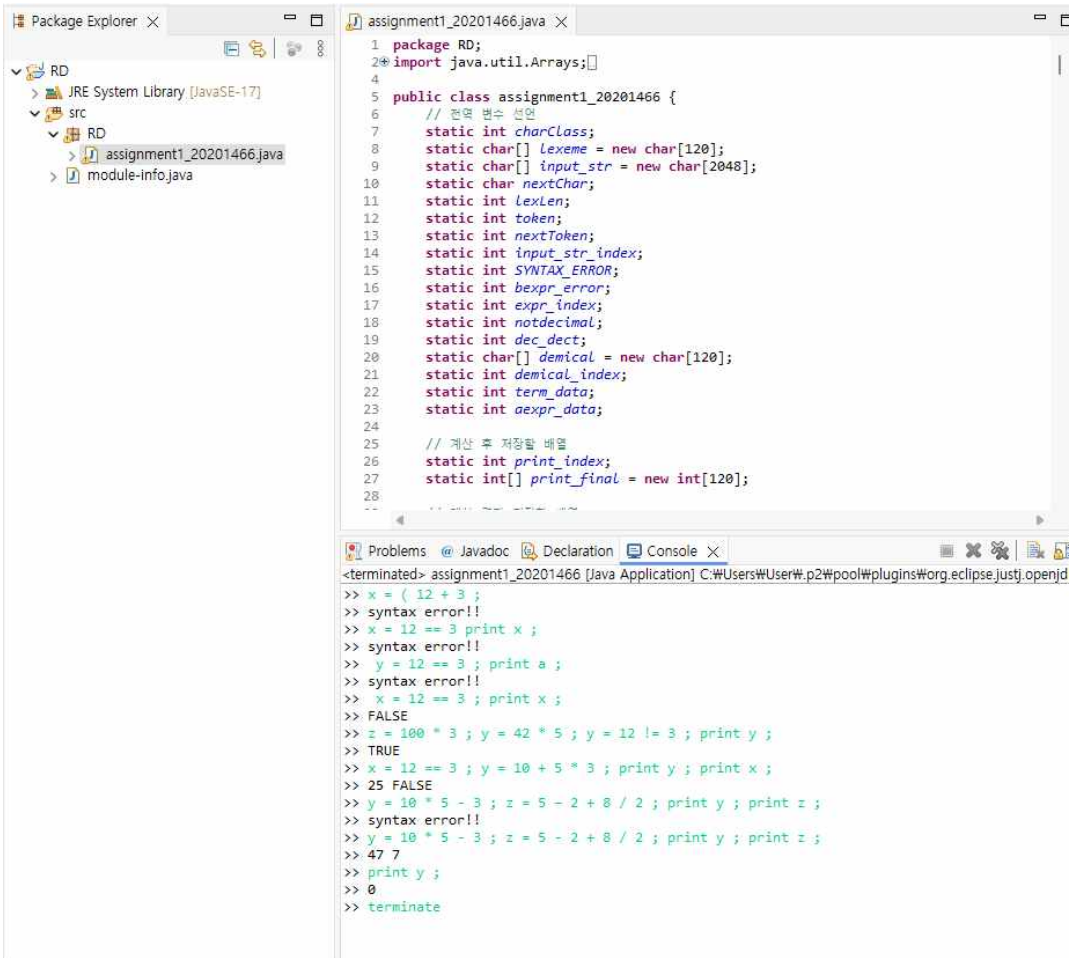
목차

1. 실행 결과

(1) C (assignment1_20201466.c)

```
/mnt/c/U/User/sa/P/homework1  main ?60   gcc assignment1_20201466.c
/mnt/c/U/User/sa/P/homework1  main ?60   ./a.out
>> x = ( 12 + 3 ;
>> syntax error!!
>> x = 12 == 3 print x ;
>> syntax error!!
>> y = 12 == 3 ; print a ;
>> syntax error!!
>>  x = 12 == 3 ; print x ;
>> FALSE
>> z = 100 * 3 ; y = 42 * 5 ; y = 12 != 3 ; print y ;
>> TRUE
>>  x = 12 == 3 ; y = 10 + 5 * 3 ; print y ; print x ;
>> 25 FALSE
>> y = 10 * 5 - 3 ; z = 5 - 2 + 8 / 2 ; print y ; print z ;
>> 47 7
>> print x ;
>> 0
>> terminate
```

(2) JAVA (assignment1_20201466.java)



(3) PYTHON (assignment1_20201466.py)

```
>>
========== RESTART: C:₩Users₩User₩pythonidle₩assignment1_20201466.py ==========
>> x = ( 12 + 3 ;
>> syntax error!!
>> x = 12 == 3 print x ;
>> syntax error!!
>> y = 12 == 3 ; print a ;
>> syntax error!!
>> x = 12 == 3 ; print x ;
>> False
>> z = 100 * 3 ; y = 42 * 5 ; y = 12 != 3 ; print y ;
>> True
>> x = 12 == 3 ; y = 10 + 5 * 3 ; print y ; print x ;
>> 25 False
>> y = 10 * 5 - 3 ; z = 5 - 2 + 8 / 2 ; print y ; print z ;
>> syntax error!!
>> y = 10 * 5 - 3 ; z = 5 - 2 + 8 / 2 ; print y ; print z ;
>> 47 7
>> print z ;
>> 0
>> terminate
>>>
```

2. 코드 설명

1) C 파일 (assignment1_20201466.c)

(1) 전역변수, 매크로 선언

```c
/* 전역 변수 선언 */
int charClass;
char lexeme[100];
char input_str[2048];
char nextChar;
int lexLen;
int token;
int nextToken;
int input_str_index;
int SYNTAX_ERROR;
int bexpr_error;
int expr_index;
int notdecimal;
int dec_dect;
char demical[100];
int demical_index;
int term_data;
int aexpr_data;

/* 계산 후 저장할 배열 */
int print_index;
int print_final[100];

/* 계산 결과 저장할 배열 */
int print_x[100];      // 1번
int print_y[100];      // 2번
int print_z[100];      // 3번

bool print_x_bool[100];   // 4번
bool print_y_bool[100];   // 5번
bool print_z_bool[100];   // 6번

/* 임시 저장공간 */
```

```
int var_x;
int var_y;
int var_z;

/* 변수(x, y, z)와 연산자 확인 변수 */
int var_detect;
int cmp_detect;

bool moving_bool;
int moving_dec;

int bool_dec;

int bool_dec_x;
int bool_dec_y;
int bool_dec_z;

/* 함수 */
/* 문자열 추출 함수 */
int lookup_op_1(char ch);
int lookup_op_2(char ch);
int lookup_var(char ch);
void addChar();
void getChar();
void getNonBlank();
int lex();

/* 파싱 함수 */
void program();
void statement();
void expr();
void bexpr();
void relop();
void aexpr();
void term();
void factor();
```

```
void number();
void dec();
void var();

/* 에러 함수 */
void error();



/* Character */
#define LETTER 1
#define DIGIT 2
#define UNKNOWN 99

/*
 * 토큰
 * ' +, -, *, /, =, !, <, >, ; '
 */
#define NULL_A 0
#define INT_LIT 10
#define IDENT 11
#define OPERATION 12

/* 연산자 */
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LESS_THAN 25
#define LESS 26
#define MORE_THAN 27
#define MORE 28
#define EQUAL 29
#define NOT_EQUAL 30
#define NOT 31
```

```
#define SEMI_COLON 32

/* 변수 */
#define VAR_X 33
#define VAR_Y 34
#define VAR_Z 35
```

프로그램의 처음에 사용할 전역변수와 매크로를 선언하여 주었습니다.

(2) 메인 함수

```
int main() {
   while (1) {
     SYNTAX_ERROR = 0;
     bexpr_error = 0;
     input_str_index = 0;
     var_x=0; var_y=0; var_z=0;
     print_index = 0;
     bool_dec = 1;
     bool_dec_x = 1; bool_dec_y = 1; bool_dec_z = 1;
     memset(input_str, '\0', sizeof(input_str));
     printf(">> ");
     fgets(input_str, 2048, stdin);

     if (strcmp(input_str, "terminate\n") == 0) {
       exit(0);
     }

     getChar();

     /*
      * RD 파싱 시작
      */
     program();

     /*
      *  출력
      */
```

```
    printf(">> ");
    if (SYNTAX_ERROR == 0) {
      int length_print = print_index;
      for (int i=0; i<print_index; i++) {
        switch (print_final[i]) {
          // INT 형 x 출력
          case 1 :
            printf("%d ", print_x[i]);
            break;

          // int 형 y 출력
          case 2 :
            printf("%d ", print_y[i]);
            break;

          // int 형 z 출력
          case 3 :
            printf("%d ", print_z[i]);
            break;

          // bool 형 x 출력
          case 4 :
            if (print_x_bool[i] == 0) {
              printf("FALSE ");
            }
            else {
              printf("TRUE ");
            }
            break;

          // bool 형 y 출력
          case 5 :
            if (print_y_bool[i] == 0) {
              printf("FALSE ");
            }
            else {
```

```
              printf("TRUE ");
            }
            break;

          // bool 형 z 출력
          case 6 :
            if (print_z_bool[i] == 0) {
              printf("FALSE ");
            }
            else {
              printf("TRUE ");
            }
            break;

          default :
            break;
        }
      }
      printf("\n");
    }
    else {
      printf("syntax error!!\n");
    }
  }
  return 0;
}
```

① 처음에 전역변수들을 초기화하여줍니다.
② input_str에 RD 파서로 판별할 문자열을 입력하여줍니다. (terminate가 들어오면 종료합니다.)
③ program()을 통해 RD 파싱을 시작합니다.
④ 파싱이 끝나고 파싱 도중 오류가 일어났다면 'syntax error!!' 메시지를 출력합니다.
⑤ 오류가 일어나지 않았다면 파싱도중 저장했던 배열(int형 x, bool형 x, int 형 y, bool형 y, int형 z, bool형 z)을 이용하여 알맞게 출력합니다.

(3) lookup_op_1

```
int lookup_op_1(char ch) {
   switch (ch) {
     case '+' :
       addChar();
       nextToken = ADD_OP;
       break;

     case '-' :
       addChar();
       nextToken = SUB_OP;
       break;

     case '*' :
       addChar();
       nextToken = MULT_OP;
       break;

     case '/' :
       addChar();
       nextToken = DIV_OP;
       break;

     case '=' :
       addChar();
       nextToken = ASSIGN_OP;
       break;

     case '!' :
       addChar();
       nextToken = NOT;
       break;

     case '<' :
       addChar();
       nextToken = LESS_THAN;
```

```
      break;

    case '>' :
      addChar();
      nextToken = MORE_THAN;
      break;

    case ';' :
      addChar();
      nextToken = SEMI_COLON;
      break;

    default :
      addChar();
      nextToken = OPERATION;
      break;
  }
  return nextToken;
}
```

① 연산자가 들어올 때 첫 번째로 판별하는 함수입니다.
② +, -, *, /, =, !, <, >, ; 로 나누어 판별합니다.

(4) lookup_op_2

```
/*
 *  연산자가 2개 이상 연속으로 올 때 자신이 쓰는
 */
int lookup_op_2(char ch) {
  switch (ch) {
    case '=' :
      // '=='
      if (nextToken == ASSIGN_OP) {
        addChar();
        nextToken = EQUAL;
      }

      // '!='
```

```
      else if (nextToken == NOT) {
        addChar();
        nextToken = NOT_EQUAL;
      }

      // '<='
      else if (nextToken == LESS_THAN) {
        addChar();
        nextToken = LESS;
      }

      // '>='
      else if (nextToken == MORE_THAN) {
        addChar();
        nextToken = MORE;
      }

      // 기타 등등
      else {
        addChar();
        nextToken = OPERATION;
      }
      break;

      // 기타 등등
    default :
      addChar();
      nextToken = OPERATION;
      break;
  }
  return nextToken;
}
```

① 두 번째부터 오는 연산자를 판별합니다.
② ==, !=, >=, <=을 판별합니다.

(5) lookup_var

```
/*
 * <var> 결과 값을 확인 하려는 함수
 */
int lookup_var(char ch) {
   switch (ch) {
     case 'x' :
       addChar();
       nextToken = VAR_X;
       break;

     case 'y' :
       addChar();
       nextToken = VAR_Y;
       break;

     case 'z' :
       addChar();
       nextToken = VAR_Z;
       break;

     default :
       addChar();
       nextToken = IDENT;
       break;
   }

   return nextToken;
}
```

① 이 토큰이 x, y, z인지 판별하는 함수입니다.

(6) addChar, getChar, getNonBlank

```
void addChar() {
   if (lexLen <= 98) {
     lexeme[lexLen++] = nextChar;
     lexeme[lexLen] = 0;
   }
   else {
     printf("error");
   }
}
/*
 *  판단할 문자가 문자인지, 숫자인지, 연산자인지 판단하는 함수
 */
void getChar() {
   if ((nextChar = input_str[input_str_index++]) != '\0') {
     if (isalpha(nextChar)) {
       charClass = LETTER;
     }
     else if (isdigit(nextChar)) {
       charClass = DIGIT;
     }
     else { charClass = UNKNOWN; }
   }
   else {
     charClass = NULL_A;
   }
}
/*
 *  공백 키를 건너 띄어주는 함수
 */
void getNonBlank() {
   while (isspace(nextChar)) {
     getChar();
   }
}
```

① addChar 함수는 지금 판별하고 있는 토큰을 배열에 저장해주는 함수입니다.
② getChar 함수는 저장해주는 문자가 문자인지, 숫자인지, 연산자인지 판별해줍니다.
③ getNonBlank 함수는 빈칸을 제거해주는 기능을 합니다.

(7) lex()

```
/*
 *  분리한 토큰의 정보를 저장해주는 함수
 */
int lex() {
   lexLen = 0;
   getNonBlank();

   switch (charClass) {
     case LETTER :
       lookup_var(nextChar);
       getChar();

       while (!isspace(nextChar)) {
         addChar();
         getChar();
         nextToken = IDENT;
       }

       break;

     case DIGIT :
       addChar();
       getChar();

       nextToken = INT_LIT;
       break;

     case UNKNOWN :
       lookup_op_1(nextChar);
       getChar();

       while (!isspace(nextChar)) {
```

```
        lookup_op_2(nextChar);
        getChar();
      }

      break;

    case NULL_A :
      nextToken = NULL_A;
      lexeme[0] = 'N';
      lexeme[1] = 'U';
      lexeme[2] = 'L';
      lexeme[3] = 'L';
      lexeme[4] = 0;
      break;
  }
//   printf("next token : %d, next lexeme : %s\n", nextToken, lexeme);
  return nextToken;
}
```

① lex 함수는 현재 토큰의 정보를 판별하여 줍니다.
② 토큰의 시작이 문자라면 공백이 나올 때까지의 문자열을 판별합니다. (lookup_var 이용)
③ 토큰의 시작이 숫자라면 숫자는 1개씩 판별합니다.
④ 토큰의 시작이 연산자라면 공백이 나올 때까지의 문자열을 판별합니다. (lookup_op_1, lookup_op_2 이용)

(8) program()

```
/*
 *  <program> -> {<statement>}
 */
void program() {
  lex();

  while (nextToken != NULL_A && SYNTAX_ERROR == 0) {
    statement();
  }
}
```

① 문자열이 끝나기 전까지 아니면 error가 나기 전까지 statement함수를 호출합니다.

(9) statement 함수

```
/*
 *  <statement> -> <var> = <expr> ; | print <var> ;
 */
void statement() {
   if (SYNTAX_ERROR == 0) {
     // <var> = <expr> ;
     if (strcmp(lexeme, "print") != 0) {
       var();

       if (SYNTAX_ERROR == 0) {
         if (nextToken != ASSIGN_OP) {
           error();
         }
         else {
           expr_index = input_str_index;
           lex();
           expr();
         }
       }

       if (SYNTAX_ERROR == 0) {
         if (nextToken != SEMI_COLON) {
           error();
         }
         else {
           lex();
         }
       }

       // <expr> 의 결과 값을 저장
       if (SYNTAX_ERROR == 0) {
         // 결과 값이 bool 형일 때
         if (bool_dec == 0) {
           switch (var_detect) {
             case VAR_X :
```

```
            bool_dec_x = 0;
            var_x = moving_bool;
            break;

        case VAR_Y :
            bool_dec_y = 0;
            var_y = moving_bool;
            break;

        case VAR_Z :
            bool_dec_z = 0;
            var_z = moving_bool;
            break;

        default :
            break;
    }
  }
  // 결과 값이 int 형일 때
  else if (bool_dec == 1){
    switch (var_detect) {
      case VAR_X :
        bool_dec_x = 1;
        var_x = moving_dec;
        break;

      case VAR_Y :
        bool_dec_y = 1;
        var_y = moving_dec;
        break;

      case VAR_Z :
        bool_dec_z = 1;
        var_z = moving_dec;
        break;
```

```
          default :
            break;
        }
      }
    }
  }
  // print <var> ;
  else {
    lex();
    var();
    if (SYNTAX_ERROR == 0) {
      if (nextToken != SEMI_COLON) {
        error();
      }
      else {
        lex();
      }
    }
    if (SYNTAX_ERROR == 0) {
      // x, y, z 구별
      switch (var_detect) {
        case VAR_X :
          // x 가 bool 형 일 때
          if (bool_dec_x == 0) {
            print_x_bool[print_index] = var_x;
            print_final[print_index++] = 4;
            print_final[print_index] = 0;
          }
          // x 가 int 형일 때
          else if (bool_dec_x == 1) {
            print_x[print_index] = var_x;
            print_final[print_index++] = 1;
            print_final[print_index] = 0;
          }
          break;
```

```
            case VAR_Y :
              // y 가 bool형 일 때
              if (bool_dec_y == 0) {
                print_y_bool[print_index] = var_y;
                print_final[print_index++] = 5;
                print_final[print_index] = 0;
              }
              // y 가 int 형 일 때
              else if (bool_dec_y == 1) {
                print_y[print_index] = var_y;
                print_final[print_index++] = 2;
                print_final[print_index] = 0;
              }
              break;
            case VAR_Z :
              // z 가 bool형 일 때
              if (bool_dec_z == 0) {
                print_z_bool[print_index] = var_z;
                print_final[print_index++] = 6;
                print_final[print_index] = 0;
              }
              // z 가 int 형일 때
              else if (bool_dec_z == 1) {
                print_z[print_index] = var_z;
                print_final[print_index++] = 3;
                print_final[print_index] = 0;
              }
              break;

            default :
              break;
          }
        }
      }
    }
}
```

① statement는 에러가 났었다면 통과되고 토큰이 print인지 아닌지로 나누어 행동됩니다.
② print 로 시작하지 않는다면 정상적으로 모두 실행 될 때(에러가 나지 않을 때), var() 함수를 호출한 후 다음 토큰이 '=' 인지 확인한 후 맞다면 lex() 함수와 expr() 함수가 호출되고 다음 토큰이 ';' 인지 판별합니다. 만약 중간에 에러가 났다면 각 과정을 통과할 수 있습니다.
그 후 expr() 함수에서 계산된 값을 임시적인 var_x, var_y, var_z에 저장합니다.
이 때 bool값인지 int형인지 판별할 수 있는 식별자의 정보도 업데이트합니다.
③ print로 시작하였다면 lex(), var() 함수를 호출하고 ';'인지 판별합니다.
그 후 var() 값에 따라 전역변수로 선언해준 배열(print_x, print_x_bool, print_y, print_y_bool, print_z, print_z_bool)에 var_x, var_y, var_z를 저장해줍니다. print_final 함수에는 어느 배열에 저장이 되어있는지 알 수 있는 판별자를 저장합니다.

(10) expr 함수

```
/*
 *  <expr> -> <bexpr> | <aexpr>
 */
void expr() {
   if (SYNTAX_ERROR == 0) {
     bexpr_error = 0;

     // 먼저 실행
     bexpr();

     // <bexpr> 이 오류가 날 시 <aexpr> 실행
     if (bexpr_error == -1) {
       SYNTAX_ERROR = 0;
       bexpr_error = 0;
       input_str_index = expr_index;
       getChar();
       lex();
       aexpr();
     }
   }
}
```

① expr 함수는 먼저 bexpr()을 실행한 후 bexpr() 과정에서 에러가 난다면 aexpr()을 실행합니다.

(11) bexpr()

```
/*
 *  <bexpr> -> <number> <relop> <number>
 */
void bexpr() {
   if (SYNTAX_ERROR == 0) {
     number();
     int front = atoi(demical);
     demical[0] = 0;

     if (SYNTAX_ERROR == 0) {
       relop();
     }

     if (SYNTAX_ERROR == 0) {
       number();
     }

     // 숫자를 변환하고 연산자의 판별을 통해 계산 (bool 형)
     if (SYNTAX_ERROR == 0) {
       int back = atoi(demical);
       demical[0] = 0;

       int bool_bexpr;

       switch (cmp_detect) {
         case EQUAL :
           bool_bexpr = (front == back);
           break;

         case NOT_EQUAL :
           bool_bexpr = (front != back);
           break;

         case MORE :
           bool_bexpr = (front >= back);
```

```
            break;

          case MORE_THAN :
            bool_bexpr = (front > back);
            break;

          case LESS :
            bool_bexpr = (front <= back);
            break;

          case LESS_THAN :
            bool_bexpr = (front < back);
            break;

          default :
            break;
        }
        // 전역 변수에 저장
        moving_bool = bool_bexpr;
        bool_dec = 0;
      }
    }
}
```

① bexpr 함수는 number, relop, number 함수를 순차적으로 호출합니다.
② 첫 number가 끝났을 때는 결과값을 front에 저장하고 두 번째 number가 끝났을 때는 결과값을 back에 저장한 다음, relop의 결과 값인 cmp_detect로 boolean값을 판별하여 bool_bexpr 에 저장합니다.

(12) relop

```
void relop() {
  if (SYNTAX_ERROR == 0) {
    switch (nextToken) {
      case EQUAL :
        cmp_detect = EQUAL;
        break;
```

```
      case NOT_EQUAL :
        cmp_detect = NOT_EQUAL;
        break;

      case MORE :
        cmp_detect = MORE;
        break;

      case MORE_THAN :
        cmp_detect = MORE_THAN;
        break;

      case LESS :
        cmp_detect = LESS;
        break;

      case LESS_THAN :
        cmp_detect = LESS_THAN;
        break;

      default :
        error();
        bexpr_error = -1;
        // lex();
        break;
    }
    if (SYNTAX_ERROR == 0) {
      lex();
    }
  }
}
```

① ==, !=, >, >=, <=, <을 판별하여 cmp_detect에 정보를 저장합니다.

(13) aexpr

```
/*
 *  <aexpr> -> <term> {(+ | -) <term>}
 */
void aexpr() {
   int front=0, back=0, operand=0;
   if (SYNTAX_ERROR == 0) {
     term();
     front = term_data;

     // 에러가 나지 않았거나 +, - 가 오면 반복
     while ((nextToken == ADD_OP || nextToken == SUB_OP) && SYNTAX_ERROR==0) {
       if (nextToken == ADD_OP) {
         operand = ADD_OP;
       }
       else if (nextToken == SUB_OP) {
         operand = SUB_OP;
       }
       lex();
       term();

       // 계산
       back = term_data;
       if (operand == ADD_OP) {
         front = front + back;
       }
       else if (operand == SUB_OP) {
         front = front - back;
       }
     }
   }
   if (SYNTAX_ERROR == 0) {
     bool_dec = 1;
     moving_dec = front;
   }
}
```

① aexpr 함수는 term() 함수를 호출하고 다음 토큰이 + 또는 -이거나 에러가 나지 않았다면 다시 term() 함수를 호출합니다.
② term 함수의 결과값은 각각 front와 back에 저장하고 판단한 연산자로 도출된 겨로가 값을 moving_dec에 저장합니다. 또한 이 값이 int형이라는 것을 판별할 판별자의 정보도 업데이트합니다.

(14) term

```
void term() {
   int front=0, back=0, operand;
   if (SYNTAX_ERROR == 0) {
     factor();
     front = atoi(demical);

     // 에러가 나지 않았거나 *, / 가 오면 반복
     while ((nextToken == MULT_OP || nextToken == DIV_OP) && SYNTAX_ERROR==0) {
       if (nextToken == MULT_OP) {
         operand = MULT_OP;
       }
       else if (nextToken == DIV_OP) {
         operand = DIV_OP;
       }
       lex();
       factor();
       // 계산
       back = atoi(demical);
       if (operand == MULT_OP) {
         front = front * back;
       }
       else if (operand == DIV_OP) {
         front = front / back;
       }
     }
   }
   if (SYNTAX_ERROR == 0) {
     term_data = front;
   }
}
```

① term 함수는 factor() 함수를 호출하고 다음 토큰이 * 또는 /이거나 에러가 나지 않았다면 다시 factor() 함수를 호출합니다.
② term 함수의 결과값은 각각 front와 back에 저장하고 판단한 연산자로 도출된 결과 값을 term_data에 저장합니다.

(15) factor

```
/*
 *  <factor> -> <number>
 */
void factor() {
   if (SYNTAX_ERROR == 0) {
     number();
   }
}
```

① number 함수를 호출합니다.

(16) number

```
/*
 *  <number> -> <dec>{<dec>}
 */
void number() {
   dec_dect = 0;
   memset(demical, '\0', sizeof(demical));
   if (SYNTAX_ERROR == 0) {
     dec();
     dec_dect++;

     // 정수이거나 에러가 나지 않았을 때
     while (notdecimal==0 && SYNTAX_ERROR == 0) {
       dec();
     }
     notdecimal = 0;
     demical_index = 0;
   }
}
```

① 정수를 저장할 demical 함수를 초기화하고 dec() 함수를 호출합니다.
② 에러가 나지 않았거나 정수가 맞다면 계속해서 dec() 함수를 호출합니다.
③ 이후 notdecimal 과 demical_index를 초기화합니다.

(17) dec

```
/*
 * <dec> -> 0|1|2|3|4|5|6|7|8|9
 */
void dec() {
  if (SYNTAX_ERROR == 0) {
    if (nextToken == INT_LIT) {
      demical[demical_index++] = lexeme[0];
      lex();
    }
    // 첫번째 <dec> 에 정수가 아니면 에러
    else if (dec_dect == 0) {
      error();
      bexpr_error = -1;
    }
    // 이후의 <dec>은 지금 판별하진 않음
    else {
      notdecimal = -1;
    }
  }
}
```

① dec 함수는 demical 배열에 차례대로 값을 저장합니다.
② 만약 첫 번째 dec함수일 때 정수가 아니라면 에러가 납니다. bexpr에 관련하여 오류인 것일 수도 있어 bexpr_error의 값도 바꿔줍니다.
③ 두 번째 이후의 dec함수가 정수가 아니라면 이후에 판별을 합니다.

(18) var 함수

```
void var() {
  if (SYNTAX_ERROR == 0) {
    if (nextToken == VAR_X || nextToken == VAR_Y || nextToken == VAR_Z) {
      if (nextToken == VAR_X) {
        var_detect = VAR_X;
      }
```

```
      else if (nextToken == VAR_Y) {
        var_detect = VAR_Y;
      }
      else if (nextToken == VAR_Z) {
        var_detect = VAR_Z;
      }
      lex();
    }
    else {
      error();
    }
  }
}
```

① var() 함수는 x, y, z인지를 판별하여 var_detect의 값을 업데이트합니다.
② x, y, z가 아니라면 에러를 호출합니다.

(19) error

```
void error() {
  SYNTAX_ERROR = -1;
}
```

① SYNTAX_ERROR의 값을 -1로 바꾸어줍니다.

(2) JAVA 파일 (assignment1_20201466.java)

(1) 전역변수, 매크로 선언

```
package RD;

import java.util.Arrays;
import java.util.Scanner;

public class assignment1_20201466 {
    // 전역변수선언
    static int charClass
    static char[] lexeme = new char[120];
    static char[] input_str = new char[2048];
    static char nextChar
    static int lexLen
    static int token
    static int nextToken
    static int input_str_index
    static int SYNTAX_ERROR
    static int bexpr_error
    static int expr_index
    static int notdecimal
    static int dec_dect
    static char[] demical = new char[120];
    static int demical_index
    static int term_data
    static int aexpr_data

    // 계산후저장할배열
    static int print_index
    static int[] print_final = new int[120];

    // 계산결과저장할배열
    static int[] print_x = new int[120];            // 1번
    static int[] print_y = new int[120];            // 2번
    static int[] print_z = new int[120];            // 3번

    static boolean[] print_x_bool = new boolean[120];     // 4번
    static boolean[] print_y_bool = new boolean[120];     // 5번
    static boolean[] print_z_bool = new boolean[120];     // 6번

    // 임시저장공간
    static int var_x
    static int var_y
    static int var_z

    // 변수(x, y, z)와연산자확인변수
    static int var_detect
    static int cmp_detect

    static boolean moving_bool
    static int moving_dec

    static int bool_dec

    static int bool_dec_x
    static int bool_dec_y
    static int bool_dec_z

    // 매크로지정
    public class TokenConstants {
        public static final int LETTER = 1;
        public static final int DIGIT = 2;
        public static final int UNKNOWN = 99;
```

```
        // 토큰
        // ' +, -, *, /, =, !, <, >, ; '
        public static final int NULL = 0;
        public static final int INT_LIT = 10;
        public static final int IDENT = 11;
        public static final int OPERATION = 12;

        // 연산자
        public static final int ASSIGN_OP = 20;
        public static final int ADD_OP = 21;
        public static final int SUB_OP = 22;
        public static final int MULT_OP = 23;
        public static final int DIV_OP = 24;
        public static final int LESS_THAN = 25;
        public static final int LESS = 26;
        public static final int MORE_THAN = 27;
        public static final int MORE = 28;
        public static final int EQUAL = 29;
        public static final int NOT_EQUAL = 30;
        public static final int NOT = 31;
        public static final int SEMI_COLON = 32;

        // 변수
        public static final int VAR_X = 33;
        public static final int VAR_Y = 34;
        public static final int VAR_Z = 35;
    }
```

① 이클립스를 사용방법을 잘 몰라 실행이 안 되어서 방법을 찾다가 패키지를 RD로 두고 'package RD;'를 추가하니 실행이 되었습니다.

② 코드에서 쓰이는 전역변수들을 선언해주었고 상수는 class를 통해 선언해 주었습니다.

(2) main 함수

```
public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        while (true) {
            SYNTAX_ERROR = 0;
            bexpr_error = 0;
            input_str_index = 0;
            var_x = 0;
            var_y = 0;
            var_z = 0;
            print_index = 0;
            bool_dec = 1;
            bool_dec_x = 1;
            bool_dec_y = 1;
            bool_dec_z = 1;
            nextToken = -1;
            Arrays.fill(input_str, '\0');

            System.out.print(">> ");
            String input = scanner.nextLine();
            input += '\0'
            input_str = input.toCharArray();

            if (input.equals("terminate\0")) {
                System.exit(0);
            }

            getChar();

            /*
```

```
            * RD 파싱시작
            */
            program();

            /*
            *  출력
            */
            System.out.print(">> ");
            if (SYNTAX_ERROR == 0) {
                int length_print = print_index
                for (int i = 0; i < print_index i++) {
                    switch (print_final[i]) {
                        // INT 형x 출력
                        case 1:
                            System.out.print(print_x[i] + " ");
                            break

                        // int 형y 출력
                        case 2:
                            System.out.print(print_y[i] + " ");
                            break

                        // int 형z 출력
                        case 3:
                            System.out.print(print_z[i] + " ");
                            break

                        // bool 형x 출력
                        case 4:
                            if (!print_x_bool[i]) {
                                System.out.print("FALSE ");
                            } else {
                                System.out.print("TRUE ");
                            }
                            break

                        // bool 형y 출력
                        case 5:
                            if (!print_y_bool[i]) {
                                System.out.print("FALSE ");
                            } else {
                                System.out.print("TRUE ");
                            }
                            break

                        // bool 형z 출력
                        case 6:
                            if (!print_z_bool[i]) {
                                System.out.print("FALSE ");
                            } else {
                                System.out.print("TRUE ");
                            }
                            break

                        default:
                            break
                    }
                }
                System.out.println();
            } else {
                System.out.println("syntax error!!");
            }
        }
    }
```

C 파일과 기능적으로 동일합니다.
① 처음에 전역변수들을 초기화하여줍니다.
② input_str에 RD 파서로 판별할 문자열을 입력하여줍니다. (terminate가 들어오면 종료합니다.)
③ program()을 통해 RD 파싱을 시작합니다.
④ 파싱이 끝나고 파싱 도중 오류가 일어났었다면 'syntax error!!' 메시지를 출력합니다.
⑤ 오류가 일어나지 않았다면 파싱도중 저장했던 배열(int형 x, bool형 x, int 형 y, bool형 y, int형 z, bool형 z)을 이용하여 알맞게 출력합니다.

(3) lookup_op_1

```
// 문자열추출함수첫번째
    static int lookup_op_1(char ch) {
        switch (ch) {
            case '+' :
                addChar();
                nextToken = TokenConstants.ADD_OP
                break

            case '-' :
                addChar();
                nextToken = TokenConstants.SUB_OP
                break

            case '*' :
                addChar();
                nextToken = TokenConstants.MULT_OP
                break

            case '/' :
                addChar();
                nextToken = TokenConstants.DIV_OP
                break

            case '=' :
                addChar();
                nextToken = TokenConstants.ASSIGN_OP
                break

            case '!' :
                addChar();
                nextToken = TokenConstants.NOT
                break

            case '<' :
                addChar();
                nextToken = TokenConstants.LESS_THAN
                break

            case '>' :
                addChar();
                nextToken = TokenConstants.MORE_THAN
                break

            case ';' :
                addChar();
                nextToken = TokenConstants.SEMI_COLON
                break

            default :
                addChar();
```

```
                nextToken = TokenConstants.OPERATION
                break
        }
        return nextToken
    }
```

C파일과 기능적으로 동일합니다.

① 연산자가 들어올 때 첫 번째로 판별하는 함수입니다.

② +, -, *, /, =, !, <, >, ; 로 나누어 판별합니다.

③ 상수 참조 시 사용한 class이름을 통해 참조합니다.

(4) lookup_op_2

```
// 문자열추출함수두번째이상
    static int lookup_op_2(char ch) {
        switch (ch) {
            case '=' :
                // '=='
                if (nextToken == TokenConstants.ASSIGN_OP) {
                    addChar();
                    nextToken = TokenConstants.EQUAL
                }

                // '!='
                else if (nextToken == TokenConstants.NOT) {
                    addChar();
                    nextToken = TokenConstants.NOT_EQUAL
                }

                // '<='
                else if (nextToken == TokenConstants.LESS_THAN) {
                    addChar();
                    nextToken = TokenConstants.LESS
                }

                // '>='
                else if (nextToken == TokenConstants.MORE_THAN) {
                    addChar();
                    nextToken = TokenConstants.MORE
                }

                // 기타등등
                else {
                    addChar();
                    nextToken = TokenConstants.OPERATION
                }
                break

            // 기타등등
            default :
                addChar();
                nextToken = TokenConstants.OPERATION
                break
        }
        return nextToken
    }
```

C파일과 기능적으로 동일합니다.

① 두 번째부터 오는 연산자를 판별합니다.

② ==, !=, >=, <=을 판별합니다.

③ 상수 참조 시 사용한 class이름을 통해 참조합니다.

(5) lookup_var

```
// x, y, z 구분함수
    static int lookup_var(char ch) {
        switch (ch) {
            case 'x' :
                addChar();
                nextToken = TokenConstants.VAR_X
                break

            case 'y' :
                addChar();
                nextToken = TokenConstants.VAR_Y
                break

            case 'z' :
                addChar();
                nextToken = TokenConstants.VAR_Z
                break

            default :
                addChar();
                nextToken = TokenConstants.IDENT
                break
        }

        return nextToken
    }
```

C파일과 기능적으로 동일합니다.

① 이 토큰이 x, y, z인지 판별하는 함수입니다.

② 상수 참조 시 사용한 class이름을 통해 참조합니다.

(6) addChar, getChar, getNonBlank

```
// 임시버퍼에토큰을저장
    static void addChar() {
        if (lexLen <= 98) {
            lexeme[lexLen++] = nextChar
            lexeme[lexLen] = '\0'
        }
        else {
            System.out.println("error");
        }
    }

    // 토큰으로나눌때까지한글자씩읽기
    static void getChar() {
        if ((nextChar = input_str[input_str_index++]) != '\0') {
            if (Character.isLetter(nextChar)) {
                charClass = TokenConstants.LETTER
            }
            else if (Character.isDigit(nextChar)) {
                charClass = TokenConstants.DIGIT
            }
            else {
                charClass = TokenConstants.UNKNOWN
            }
        }
        else {
```

```
            charClass = TokenConstants.NULL
        }
    }

    // 공백빼기
    static void getNonBlank() {
        while (Character.isWhitespace(nextChar)) {
            getChar();
        }
    }
```

C 코드와 기능적으로 동일합니다. 함수의 변동이 있습니다.

① addChar 함수는 지금 판별하고 있는 토큰을 배열에 저장해주는 함수입니다.

② getChar 함수는 저장해주는 문자가 문자인지, 숫자인지, 연산자인지 판별해줍니다.

③ getNonBlank 함수는 빈칸을 제거해주는 기능을 합니다.

④ 상수 참조 시 사용한 class이름을 통해 참조합니다.

(7) lex

```
// 토큰에대한정보를저장
    static int lex() {
        lexLen = 0;
        Arrays.fill(lexeme, '\0');
        getNonBlank();

        switch (charClass) {
            case TokenConstants.LETTER:
                lookup_var(nextChar);
                getChar();

                while (nextChar!='\0' && !Character.isWhitespace(nextChar)) {
                    addChar();
                    getChar();
                    nextToken = TokenConstants.IDENT
                }

                break

            case TokenConstants.DIGIT:
                addChar();
                getChar();

                nextToken = TokenConstants.INT_LIT
                break

            case TokenConstants.UNKNOWN:
                lookup_op_1(nextChar);
                getChar();

                while (nextChar!='\0' && !Character.isWhitespace(nextChar)) {
                    lookup_op_2(nextChar);
                    getChar();
                }

                break

            case TokenConstants.NULL:
                nextToken = TokenConstants.NULL
                lexeme[0] = 'N'
                lexeme[1] = 'U'
```

```
                lexeme[2] = 'L'
                lexeme[3] = 'L'
                lexeme[4] = '\0'
                break
        }
//          String str = new String(lexeme).replaceAll("\0", "");
//          System.out.printf("next token : %d, next lexeme : %s\n", nextToken, str);
        return nextToken
    }
```

C 코드와 기능적으로 동일합니다.

① lex 함수는 현재 토큰의 정보를 판별하여 줍니다.

② 토큰의 시작이 문자라면 공백이 나올 때까지의 문자열을 판별합니다. (lookup_var 이용)

③ 토큰의 시작이 숫자라면 숫자는 1개씩 판별합니다.

④ 토큰의 시작이 연산자라면 공백이 나올 때까지의 문자열을 판별합니다. (lookup_op_1, lookup_op_2 이용)

⑤ 반복 시 isWhitespace의 구분하는 문자에는 널문자가 들어가지 않기 때문에 그것을 판별해주기 위해서 nextChar != '\0' 코드가 들어갑니다.

(8) program

```
// 파싱함수
    /*
     *  <program> -> {<statement>}
     */
    static void program() {
        lex();

        while (nextToken != TokenConstants.NULL && SYNTAX_ERROR == 0) {
            statement();
        }
    }
```

C 코드와 기능적으로 동일합니다.

① 문자열이 끝나기 전까지 아니면 error가 나기 전까지 statement함수를 호출합니다.

(9) statement

```
    /*
     *  <statement> -> <var> = <expr> ; | print <var> ;
     */
    static void statement() {
        String str = new String(lexeme).replaceAll("\0", "");
        if (SYNTAX_ERROR == 0) {
            // <var> = <expr> ;
            if (!"print".equals(str)) {
                var();

                if (SYNTAX_ERROR == 0) {
                    if (nextToken != TokenConstants.ASSIGN_OP) {
                        error();
                    } else {
                        expr_index = input_str_index
                        lex();
                        expr();
                    }
                }

                if (SYNTAX_ERROR == 0) {
                    if (nextToken != TokenConstants.SEMI_COLON) {
```

```
            error();
        } else {
            lex();
        }
    }

    // <expr> 의결과값을저장
    if (SYNTAX_ERROR == 0) {
        // 결과값이bool 형일때
        if (bool_dec == 0) {
            switch (var_detect) {
                case TokenConstants.VAR_X:
                    bool_dec_x = 0;
                    var_x = moving_bool ? 1 : 0;
                    break

                case TokenConstants.VAR_Y:
                    bool_dec_y = 0;
                    var_y = moving_bool ? 1 : 0;
                    break

                case TokenConstants.VAR_Z:
                    bool_dec_z = 0;
                    var_z = moving_bool ? 1 : 0;
                    break

                default:
                    break
            }
        }
        // 결과값이int 형일때
        else if (bool_dec == 1) {
            switch (var_detect) {
                case TokenConstants.VAR_X:
                    bool_dec_x = 1;
                    var_x = moving_dec
                    break

                case TokenConstants.VAR_Y:
                    bool_dec_y = 1;
                    var_y = moving_dec
                    break

                case TokenConstants.VAR_Z:
                    bool_dec_z = 1;
                    var_z = moving_dec
                    break

                default:
                    break
            }
        }
    }
}
// print <var> ;
else {
    lex();
    var();

    if (SYNTAX_ERROR == 0) {
        if (nextToken != TokenConstants.SEMI_COLON) {
            error();
        } else {
            lex();
        }
```

```
                }

                if (SYNTAX_ERROR == 0) {
                    // x, y, z 구별
                    switch (var_detect) {
                        case TokenConstants.VAR_X:
                            // x 가bool 형일때
                            if (bool_dec_x == 0) {
                                print_x_bool[print_index] = var_x != 0;
                                print_final[print_index++] = 4;
                                print_final[print_index] = 0;
                            }
                            // x 가int 형일때
                            else if (bool_dec_x == 1) {
                                print_x[print_index] = var_x
                                print_final[print_index++] = 1;
                                print_final[print_index] = 0;
                            }
                            break

                        case TokenConstants.VAR_Y:
                            // y 가bool형 일때
                            if (bool_dec_y == 0) {
                                print_y_bool[print_index] = var_y != 0;
                                print_final[print_index++] = 5;
                                print_final[print_index] = 0;
                            }
                            // y 가int 형일때
                            else if (bool_dec_y == 1) {
                                print_y[print_index] = var_y
                                print_final[print_index++] = 2;
                                print_final[print_index] = 0;
                            }
                            break

                        case TokenConstants.VAR_Z:
                            // z 가bool형 일때
                            if (bool_dec_z == 0) {
                                print_z_bool[print_index] = var_z != 0;
                                print_final[print_index++] = 6;
                                print_final[print_index] = 0;
                            }
                            // z 가int 형일때
                            else if (bool_dec_z == 1) {
                                print_z[print_index] = var_z
                                print_final[print_index++] = 3;
                                print_final[print_index] = 0;
                            }
                            break

                        default:
                            break
                    }
                }
            }
        }
    }
```

C 코드와 기능적으로 동일합니다.
① statement는 에러가 났었다면 통과되고 토큰이 print인지 아닌지로 나누어 행동됩니다.
② print 로 시작하지 않는다면 정상적으로 모두 실행 될 때(에러가 나지 않을 때), var() 함수를 호출한 후 다음 토큰이 '=' 인지 확인한 후 맞다면 lex() 함수와 expr() 함수가 호출되고 다음 토큰이 ';' 인지 판별합니다. 만약 중간에 에러가 났다면 각 과정을 통과할 수 있습니다.

그 후 expr() 함수에서 계산된 값을 임시적인 var_x, var_y, var_z에 저장합니다.
이 때 bool값인지 int형인지 판별할 수 있는 식별자의 정보도 업데이트합니다.
③ print로 시작하였다면 lex(), var() 함수를 호출하고 ';'인지 판별합니다.
그 후 var() 값에 따라 전역변수로 선언해준 배열(print_x, print_x_bool, print_y, print_y_bool, print_z, print_z_bool)에 var_x, var_y, var_z를 저장해줍니다. print_final 함수에는 어느 배열에 저장이 되어있는지 알 수 있는 판별자를 저장합니다.

(10) expr

```
/*
 *  <expr> -> <bexpr> | <aexpr>
 */
static void expr() {
    if (SYNTAX_ERROR == 0) {
        bexpr_error = 0;

        // 먼저실행
        bexpr();

        // <bexpr> 이오류가날시<aexpr> 실행
        if (bexpr_error == -1) {
            SYNTAX_ERROR = 0;
            bexpr_error = 0;
            input_str_index = expr_index
            getChar();
            lex();
            aexpr();
        }
    }
}
```

C 코드와 기능적으로 동일합니다.
① expr 함수는 먼저 bexpr()을 실행한 후 bexpr() 과정에서 에러가 난다면 aexpr()을 실행합니다.

(11) bexpr

```
/*
 *  <bexpr> -> <number> <relop> <number>
 */
static void bexpr() {
    int front=0, back=0;
    if (SYNTAX_ERROR == 0) {
        number();
        if (demical[0] != '\0') {
            String str_front = new String(demical).replaceAll("\0", "");
            front = Integer.parseInt(str_front);
        }
        else {
            error();
        }


        if (SYNTAX_ERROR == 0) {
            relop();
        }

        if (SYNTAX_ERROR == 0) {
            number();
        }
```

```
            // 숫자를변환하고연산자의판별을통해계산(bool 형)
            if (SYNTAX_ERROR == 0) {
                if (demical[0] != '\0') {
                    String str_back = new String(demical).replaceAll("\0", "");
                    back = Integer.parseInt(str_back);
                }
                else {
                    error();
                }

                boolean bool_bexpr = false

                switch (cmp_detect) {
                    case TokenConstants.EQUAL:
                        bool_bexpr = (front == back);
                        break

                    case TokenConstants.NOT_EQUAL:
                        bool_bexpr = (front != back);
                        break

                    case TokenConstants.MORE:
                        bool_bexpr = (front >= back);
                        break

                    case TokenConstants.MORE_THAN:
                        bool_bexpr = (front > back);
                        break

                    case TokenConstants.LESS:
                        bool_bexpr = (front <= back);
                        break

                    case TokenConstants.LESS_THAN:
                        bool_bexpr = (front < back);
                        break

                    default:
                        break
                }
                // 전역변수에저장
                moving_bool = bool_bexpr
                bool_dec = 0;
            }
        }
    }
```

C 코드와 기능적으로 동일합니다.

① bexpr 함수는 number, relop, number 함수를 순차적으로 호출합니다.

② 첫 number가 끝났을 때는 결과값을 front에 저장하고 두 번째 number가 끝났을 때는 결과값을 back에 저장한 다음, relop의 결과 값인 cmp_detect로 boolean값을 판별하여 bool_bexpr 에 저장합니다.

④ char 배열을 스트링으로 바꿔주는 작업이 추가되었습니다.

(12) relop

```
    /*
     *  <relop> -> == | != | < | > | <= | >=
     */
    static void relop() {
```

```
        if (SYNTAX_ERROR == 0) {
            switch (nextToken) {
                case TokenConstants.EQUAL:
                    cmp_detect = TokenConstants.EQUAL
                    break

                case TokenConstants.NOT_EQUAL:
                    cmp_detect = TokenConstants.NOT_EQUAL
                    break

                case TokenConstants.MORE:
                    cmp_detect = TokenConstants.MORE
                    break

                case TokenConstants.MORE_THAN:
                    cmp_detect = TokenConstants.MORE_THAN
                    break

                case TokenConstants.LESS:
                    cmp_detect = TokenConstants.LESS
                    break

                case TokenConstants.LESS_THAN:
                    cmp_detect = TokenConstants.LESS_THAN
                    break

                default:
                    error();
                    bexpr_error = -1;
                    // lex();
                    break
            }
            if (SYNTAX_ERROR == 0) {
                lex();
            }
        }
    }
```

C 코드와 기능적으로 동일합니다.

① ==, !=, >, >=, <=, <을 판별하여 cmp_detect에 정보를 저장합니다.

(13) aexpr

```
    /*
     *  <aexpr> -> <term> {(+ | -) <term>}
     */
    static void aexpr() {
        int front = 0, back = 0, operand = 0;
        if (SYNTAX_ERROR == 0) {
            term();
            front = term_data

            // 에러가나지않았거나+, - 가오면반복
            while ((nextToken == TokenConstants.ADD_OP || nextToken ==
TokenConstants.SUB_OP) && SYNTAX_ERROR == 0) {
                if (nextToken == TokenConstants.ADD_OP) {
                    operand = TokenConstants.ADD_OP
                } else if (nextToken == TokenConstants.SUB_OP) {
                    operand = TokenConstants.SUB_OP
                }
                lex();
                term();
```

```
                // 계산
                back = term_data
                if (operand == TokenConstants.ADD_OP) {
                    front = front + back
                } else if (operand == TokenConstants.SUB_OP) {
                    front = front - back
                }
            }
        }
        if (SYNTAX_ERROR == 0) {
            bool_dec = 1;
            moving_dec = front
        }
    }
```

C 코드와 기능적으로 동일합니다.

① aexpr 함수는 term() 함수를 호출하고 다음 토큰이 + 또는 -이거나 에러가 나지 않았다면 다시 term() 함수를 호출합니다.

② term 함수의 결과값은 각각 front와 back에 저장하고 판단한 연산자로 도출된 겨로가 값을 moving_dec에 저장합니다. 또한 이 값이 int형이라는 것을 판별할 판별자의 정보도 업데이트합니다.

(14) term

```
    /*
     *  <term> -> <factor> {(* | /) <factor>
     */
    static void term() {
        int front = 0, back = 0, operand = 0;
        if (SYNTAX_ERROR == 0) {
            factor();
            if (demical[0] != '\0') {
                String str_front = new String(demical).replaceAll("\0", "");
                front = Integer.parseInt(str_front);
            }
            else {
                error();
            }

            // 에러가나지않았거나*, / 가오면반복
            while ((nextToken == TokenConstants.MULT_OP || nextToken ==
TokenConstants.DIV_OP) && SYNTAX_ERROR == 0) {
                if (nextToken == TokenConstants.MULT_OP) {
                    operand = TokenConstants.MULT_OP
                } else if (nextToken == TokenConstants.DIV_OP) {
                    operand = TokenConstants.DIV_OP
                }
                lex();
                factor();

                // 계산
                if (demical[0] != '\0') {
                    String str_back = new String(demical).replaceAll("\0", "");
                    back = Integer.parseInt(str_back);
                }
                else {
                    error();
                }
                if (operand == TokenConstants.MULT_OP) {
                    front = front * back
                } else if (operand == TokenConstants.DIV_OP) {
```

```
                    front = front / back
                }
            }
        }
        if (SYNTAX_ERROR == 0) {
            term_data = front
        }
    }
```

C 코드와 기능적으로 동일합니다.
① term 함수는 factor() 함수를 호출하고 다음 토큰이 * 또는 /이거나 에러가 나지 않았다면 다시 factor() 함수를 호출합니다.
② term 함수의 결과값은 각각 front와 back에 저장하고 판단한 연산자로 도출된 결과 값을 term_data에 저장합니다.
③ char 배열을 스트링으로 바꿔주는 작업이 추가되었습니다.

(15) factor

```
    /*
     *  <factor> -> <number>
     */
    static void factor() {
        if (SYNTAX_ERROR == 0) {
            number();
        }
    }
```

C 코드와 기능적으로 동일합니다.
① number 함수를 호출합니다.

(16) number

```
    /*
     *  <number> -> <dec>{<dec>}
     */
    static void number() {
        dec_dect = 0;
        Arrays.fill(demical, '\0');
        if (SYNTAX_ERROR == 0) {
            dec();
            dec_dect++;

            // 정수이거나 에러가 나지 않았을 때
            while (notdecimal == 0 && SYNTAX_ERROR == 0) {
                dec();
```

```
            }
            notdecimal = 0;
            demical_index = 0;
        }
    }
```

C 코드와 기능적으로 동일합니다.

① 정수를 저장할 demical 함수를 초기화하고 dec() 함수를 호출합니다.

② 에러가 나지 않았거나 정수가 맞다면 계속해서 dec() 함수를 호출합니다.

③ 이후 notdecimal 과 demical_index를 초기화합니다.

(17) dec

```
    /*
     * <dec> -> 0|1|2|3|4|5|6|7|8|9
     */
    static void dec() {
        if (SYNTAX_ERROR == 0) {
            if (nextToken == TokenConstants.INT_LIT) {
                demical[demical_index++] = lexeme[0];
                lex();
            }
            // 첫번째 <dec> 에 정수가 아니면 에러
            else if (dec_dect == 0) {
                error();
                bexpr_error = -1;
            }
            // 이후의 <dec>은 지금 판별하진 ㅇ낳음
            else {
                notdecimal = -1;
            }
        }
    }
```

C 코드와 기능적으로 동일합니다.

① dec 함수는 demical 배열에 차례대로 값을 저장합니다.

② 만약 첫 번째 dec함수일 때 정수가 아니라면 에러가 납니다. bexpr에 관련하여 오류인 것일 수도 있어 bexpr_error의 값도 바꿔줍니다.

③ 두 번째 이후의 dec함수가 정수가 아니라면 이후에 판별을 합니다.

(18) var

```
    /*
     *  <var> -> x | y | z
     */
    static void var() {
        if (SYNTAX_ERROR == 0) {
```

```
            if (nextToken == TokenConstants.VAR_X || nextToken ==
TokenConstants.VAR_Y || nextToken == TokenConstants.VAR_Z) {
                if (nextToken == TokenConstants.VAR_X) {
                    var_detect = TokenConstants.VAR_X;
                } else if (nextToken == TokenConstants.VAR_Y) {
                    var_detect = TokenConstants.VAR_Y;
                } else if (nextToken == TokenConstants.VAR_Z) {
                    var_detect = TokenConstants.VAR_Z;
                }
                lex();
            } else {
                error();
            }
        }
    }
```

C 코드와 기능적으로 동일

① var() 함수는 x, y, z인지를 판별하여 var_detect의 값을 업데이트합니다.

② x, y, z가 아니라면 에러를 호출합니다.

(19) error

```
    // 에러 함수
    static void error() {
        SYNTAX_ERROR = -1;
    }
```

C 코드와 기능적으로 동일합니다.

① SYNTAX_ERROR의 값을 -1로 바꾸어줍니다.

(3) PYTHON 파일 (202014661_20201466.py)

(1) 전역변수, 매크로 선언

```
# 문자 분류 상수
LETTER = 1
DIGIT = 2
UNKNOWN = 99

# 토큰 상수
NULL = 0
INT_LIT = 10
IDENT = 11
OPERATION = 12

# 연산자 상수
ASSIGN_OP = 20
ADD_OP = 21
SUB_OP = 22
MULT_OP = 23
DIV_OP = 24
LESS_THAN = 25
LESS = 26
MORE_THAN = 27
MORE = 28
EQUAL = 29
NOT_EQUAL = 30
NOT = 31
SEMI_COLON = 32

# 변수 상수
VAR_X = 33
VAR_Y = 34
VAR_Z = 35

# 전역 변수
charClass = 0
lexeme = ""
input_str = ""
nextChar = ""
lexLen = 0
token = 0
nextToken = 0
input_str_index = 0
SYNTAX_ERROR = 0
bexpr_error = 0
expr_index = 0
```

```
notdecimal = 0
dec_dect = 0
demical = ""
demical_index = 0
term_data = 0
aexpr_data = 0

# 계산 결과 저장 배열
# print_index = 0
print_final = []
print_x = []
print_y = []
print_z = []

# 임시 저장 변수
var_x = 0
var_y = 0
var_z = 0

# 변수 및 연산자 확인 변수
var_detect = 0
cmp_detect = 0

moving_bool = False
moving_dec = 0

bool_dec = 1
bool_dec_x = 1
bool_dec_y = 1
bool_dec_z = 1
```

C 코드와 기능적으로 동일합니다.
① 프로그램의 처음에 사용할 전역변수와 매크로를 선언하여 주었습니다.

(2) 메인 함수

```
if __name__ == "__main__":
    while True:
        # 초기화
        SYNTAX_ERROR = 0
        bexpr_error = 0
        input_str_index = 0
        var_x = 0
        var_y = 0
        var_z = 0
        print_index = 0
```

```
bool_dec = 1
bool_dec_x = 1
bool_dec_y = 1
bool_dec_z = 1
input_str = ""
print_x = []
print_y = []
print_z = []
print_final = []

# 사용자 입력 받기
print(">> ", end="")
input_str = input()

if input_str.strip() == "terminate":
    break

input_str = input_str + '\0'
getChar()

# RD 파싱 시작
program()

# 출력
print(">> ", end="")
if SYNTAX_ERROR == 0:
    print_x_index = 0
    print_y_index = 0
    print_z_index = 0

    for i in range(len(print_final)):
        if print_final[i] == 1:
            print(print_x[print_x_index], end=" ")
            print_x_index += 1
        elif print_final[i] == 2:
            print(print_y[print_y_index], end=" ")
            print_y_index += 1
        elif print_final[i] == 3:
            print(print_z[print_z_index], end=" ")
            print_z_index += 1
    print()
else:
    print("syntax error!!")
```

C 코드와 기능적으로 동일합니다.
① 처음에 전역변수들을 초기화하여줍니다.
② input_str에 RD 파서로 판별할 문자열을 입력하여줍니다. (terminate가 들어오면 종료합니다.)
③ program()을 통해 RD 파싱을 시작합니다.
④ 파싱이 끝나고 파싱 도중 오류가 일어났다면 'syntax error!!' 메시지를 출력합니다.
⑤ 오류가 일어나지 않았다면 파싱도중 저장했던 배열(int형 x, int 형 y, int형 z)을 이용하여 알맞게 출력합니다. bool 형 배열을 따로 만들지 않았습니다.

(3) lookup_op_1

```
# 연산자 확인 첫번째
def lookup_op_1(ch):
    global nextToken, ADD_OP, SUB_OP, MULT_OP, DIV_OP, ASSIGN_OP, NOT, LESS_THAN, MORE_THAN, SEMI_COLON, OPERATION
    if ch == '+':
        addChar()
        nextToken = ADD_OP
    elif ch == '-':
        addChar()
        nextToken = SUB_OP
    elif ch == '*':
        addChar()
        nextToken = MULT_OP
    elif ch == '/':
        addChar()
        nextToken = DIV_OP
    elif ch == '=':
        addChar()
        nextToken = ASSIGN_OP
    elif ch == '!':
        addChar()
        nextToken = NOT
    elif ch == '<':
        addChar()
        nextToken = LESS_THAN
    elif ch == '>':
        addChar()
        nextToken = MORE_THAN
    elif ch == ';':
        addChar()
        nextToken = SEMI_COLON
    else:
        addChar()
        nextToken = OPERATION
```

```
    return nextToken
```

C파일과 기능적으로 동일합니다.
① 연산자가 들어올 때 첫 번째로 판별하는 함수입니다.
② +, -, *, /, =, !, <, >, ; 로 나누어 판별합니다.

(4) lookup_op_2

```
# 연산자 확인 두번째 이후
def lookup_op_2(ch):
    global nextToken, ASSIGN_OP, EQUAL, NOT, NOT_EQUAL, LESS, LESS_THAN, MORE,
MORE_THAN, OPERATION
    if ch == '=':
        if nextToken == ASSIGN_OP:
            addChar()
            nextToken = EQUAL
        elif nextToken == NOT:
            addChar()
            nextToken = NOT_EQUAL
        elif nextToken == LESS_THAN:
            addChar()
            nextToken = LESS
        elif nextToken == MORE_THAN:
            addChar()
            nextToken = MORE
        else:
            addChar()
            nextToken = OPERATION
    else:
        addChar()
        nextToken = OPERATION
    return nextToken
```

C파일과 기능적으로 동일합니다.
① 두 번째부터 오는 연산자를 판별합니다.
② ==, !=, >=, <=을 판별합니다.
③ 상수 참조 시 사용한 class이름을 통해 참조합니다.

(5) lookup_var

```
# x, y, z 구분
def lookup_var(ch):
    global nextToken, VAR_X, VAR_Y, VAR_Z

    if ch == 'x':
        addChar()
```

```
        nextToken = VAR_X
    elif ch == 'y':
        addChar()
        nextToken = VAR_Y
    elif ch == 'z':
        addChar()
        nextToken = VAR_Z
    else:
        addChar()
        nextToken = IDENT

    return nextToken
```

C파일과 기능적으로 동일합니다.
① 이 토큰이 x, y, z인지 판별하는 함수입니다.

(6) addChar, getChar, getNonBlank

```
# 임시버퍼에 토큰 저장
def addChar():
    global lexLen, lexeme, nextChar
    lexeme = lexeme + nextChar

# 토큰이 될 때까지 문자 읽기
def getChar():
    global input_str_index, nextChar, charClass, input_str, LETTER, DIGIT,
UNKNOWN, NULL

    if input_str_index < len(input_str):
        nextChar = input_str[input_str_index]
        input_str_index += 1
        if nextChar.isalpha():
            charClass = LETTER
        elif nextChar.isdigit():
            charClass = DIGIT
        elif nextChar == '\0' :
            charClass = NULL
        else:
            charClass = UNKNOWN
    else:
        charClass = NULL

# 공백 무시
```

```
def getNonBlank():
    while nextChar.isspace():
        getChar()
```

C 코드와 기능적으로 동일합니다. 함수의 변동이 있습니다.
① addChar 함수는 지금 판별하고 있는 토큰을 배열에 저장해주는 함수입니다.
② getChar 함수는 저장해주는 문자가 문자인지, 숫자인지, 연산자인지 판별해줍니다.
③ getNonBlank 함수는 빈칸을 제거해주는 기능을 합니다.

(7) lex

```
# 토큰의 정보 저장
def lex():
    global lexLen, nextToken, lexeme, nextChar, charClass, LETTER, DIGIT,
UNKNOWN, NULL
    lexeme = ""
    lexLen = 0
    getNonBlank()

    if charClass == LETTER:

        lookup_var(nextChar)
        getChar()

        while nextChar != '\0' and not nextChar.isspace():
            addChar()
            getChar()
            nextToken = IDENT

    elif charClass == DIGIT:
        addChar()
        getChar()

        nextToken = INT_LIT

    elif charClass == UNKNOWN:
        lookup_op_1(nextChar)
        getChar()

        while nextChar != '\0' and not nextChar.isspace():
            lookup_op_2(nextChar)
            getChar()
```

```
    elif charClass == NULL:
        nextToken = NULL
        lexeme = 'NULL'

    # print(f"next token: {nextToken}, next lexeme: {lexeme}")
    return nextToken
```

C 코드와 기능적으로 동일합니다.
① lex 함수는 현재 토큰의 정보를 판별하여 줍니다.
② 토큰의 시작이 문자라면 공백이 나올 때까지의 문자열을 판별합니다. (lookup_var 이용)
③ 토큰의 시작이 숫자라면 숫자는 1개씩 판별합니다.
④ 토큰의 시작이 연산자라면 공백이 나올 때까지의 문자열을 판별합니다. (lookup_op_1, lookup_op_2 이용)
⑤ 반복 시 isWhitespace의 구분하는 문자에는 널문자가 들어가지 않기 때문에 그것을 판별해주기 위해서 nextChar != '\0' 코드가 들어갑니다.

(8) program

```
# <program> -> {<statement>}
def program():
    lex()

    while nextToken != NULL and SYNTAX_ERROR == 0:
        statement()
```

C 코드와 기능적으로 동일합니다.
① 문자열이 끝나기 전까지 아니면 error가 나기 전까지 statement함수를 호출합니다.

(9) statement

```
# <statement> -> <var> = <expr> ; | print <var> ;
def statement():
    global SYNTAX_ERROR, lexeme, nextToken, input_str_index, expr_index, bool_dec, bool_dec_x, bool_dec_y, bool_dec_z, var_detect, var_x, var_y, var_z, VAR_X, VAR_Y, VAR_Z
    global print_x, print_y, print_z, print_x_bool, print_y_bool, print_z_bool, print_index, print_final, moving_bool, moving_dec
    if SYNTAX_ERROR == 0:
        # <var> = <expr> ;
        if lexeme != "print":
```

```
    var()

    if SYNTAX_ERROR == 0:
        if nextToken != ASSIGN_OP:
            error()
        else:
            expr_index = input_str_index
            lex()
            expr()

    if SYNTAX_ERROR == 0:
        if nextToken != SEMI_COLON:
            error()
        else:
            lex()

    # <expr> 의 결과 값을 저장
    if SYNTAX_ERROR == 0:
        # 결과 값이 bool 형일 때
        if bool_dec == 0:
            if var_detect == VAR_X:
                bool_dec_x = 0
                var_x = moving_bool
            elif var_detect == VAR_Y:
                bool_dec_y = 0
                var_y = moving_bool
            elif var_detect == VAR_Z:
                bool_dec_z = 0
                var_z = moving_bool
        # 결과 값이 int 형일 때
        elif bool_dec == 1:
            if var_detect == VAR_X:
                bool_dec_x = 1
                var_x = moving_dec
            elif var_detect == VAR_Y:
                bool_dec_y = 1
                var_y = moving_dec
            elif var_detect == VAR_Z:
                bool_dec_z = 1
                var_z = moving_dec
# print <var> ;
```

```
        else:
            lex()
            var()

            if SYNTAX_ERROR == 0:
                if nextToken != SEMI_COLON:
                    error()
                else:
                    lex()

            if SYNTAX_ERROR == 0:
                # x, y, z 구별
                if var_detect == VAR_X:
                    print_x.append(var_x)
                    print_final.append(1)

                elif var_detect == VAR_Y:
                    print_y.append(var_y)
                    print_final.append(2)

                elif var_detect == VAR_Z :
                    print_z.append(var_z)
                    print_final.append(3)
```

C 코드와 기능적으로 동일합니다.
① statement는 에러가 났었다면 통과되고 토큰이 print인지 아닌지로 나누어 행동됩니다.
② print 로 시작하지 않는다면 정상적으로 모두 실행 될 때(에러가 나지 않을 때), var() 함수를 호출한 후 다음 토큰이 '=' 인지 확인한 후 맞다면 lex() 함수와 expr() 함수가 호출되고 다음 토큰이 ';' 인지 판별합니다. 만약 중간에 에러가 났다면 각 과정을 통과할 수 있습니다.
그 후 expr() 함수에서 계산된 값을 임시적인 var_x, var_y, var_z에 저장합니다.
이 때 bool값인지 int형인지 판별할 수 있는 식별자의 정보도 업데이트합니다.
③ print로 시작하였다면 lex(), var() 함수를 호출하고 ';'인지 판별합니다.
그 후 var() 값에 따라 전역변수로 선언해준 배열(print_x, print_y, print_z)에 var_x, var_y, var_z를 저장해줍니다. print_final함수에는 어느 배열에 저장이 되어있는지 알 수 있는 판별자를 저장합니다.

(10) expr

```
# <expr> -> <bexpr> | <aexpr>
def expr():
    global SYNTAX_ERROR, bexpr_error, expr_index, input_str_index
```

```
    if SYNTAX_ERROR == 0:
        bexpr_error = 0

        # 먼저 실행
        bexpr()

        # <bexpr> 이 오류가 날 시 <aexpr> 실행
        if bexpr_error == -1:
            SYNTAX_ERROR = 0
            bexpr_error = 0
            input_str_index = expr_index
            getChar()
            lex()
            aexpr()
```

C 코드와 기능적으로 동일합니다.
① expr 함수는 먼저 bexpr()을 실행한 후 bexpr() 과정에서 에러가 난다면 aexpr()을 실행합니다.

(11) bexpr

```
# <bexpr> -> <number> <relop> <number>
def bexpr():
    global SYNTAX_ERROR, demical, cmp_detect, moving_bool, bool_dec

    if SYNTAX_ERROR == 0:
        number()
        if (len(demical)==0)  :
            error()
        else :
            front = int(demical)

        if SYNTAX_ERROR == 0:
            relop()

        if SYNTAX_ERROR == 0:
            number()

        # 숫자를 변환하고 연산자의 판별을 통해 계산 (bool 형)
        if SYNTAX_ERROR == 0:
            if (len(demical) == 0):
```

```
            error()
        else :
            back = int(demical)

        bool_bexpr = 0

        if cmp_detect == EQUAL:
            bool_bexpr = (front == back)
        elif cmp_detect == NOT_EQUAL:
            bool_bexpr = (front != back)
        elif cmp_detect == MORE:
            bool_bexpr = (front >= back)
        elif cmp_detect == MORE_THAN:
            bool_bexpr = (front > back)
        elif cmp_detect == LESS:
            bool_bexpr = (front <= back)
        elif cmp_detect == LESS_THAN:
            bool_bexpr = (front < back)

        # 전역 변수에 저장
        moving_bool = bool_bexpr
        bool_dec = 0
```

C 코드와 기능적으로 동일합니다.
① bexpr 함수는 number, relop, number 함수를 순차적으로 호출합니다.
② 첫 number가 끝났을 때는 결과값을 front에 저장하고 두 번째 number가 끝났을 때는 결과값을 back에 저장한 다음, relop의 결과 값인 cmp_detect로 boolean값을 판별하여 bool_bexpr 에 저장합니다.

(12) relop

```
# <relop> -> == | != | < | > | <= | >=
def relop():
    global SYNTAX_ERROR, nextToken, cmp_detect, bexpr_error

    if SYNTAX_ERROR == 0:
        if nextToken == EQUAL:
            cmp_detect = EQUAL
        elif nextToken == NOT_EQUAL:
            cmp_detect = NOT_EQUAL
        elif nextToken == MORE:
```

```
            cmp_detect = MORE
        elif nextToken == MORE_THAN:
            cmp_detect = MORE_THAN
        elif nextToken == LESS:
            cmp_detect = LESS
        elif nextToken == LESS_THAN:
            cmp_detect = LESS_THAN
        else:
            error()
            bexpr_error = -1

        if SYNTAX_ERROR == 0:
            lex()
```

C 코드와 기능적으로 동일합니다.
① ==, !=, >, >=, <=, <을 판별하여 cmp_detect에 정보를 저장합니다.

(13) aexpr

```
# <aexpr> -> <term> {(+ | -) <term>}
def aexpr():
    global SYNTAX_ERROR, nextToken, term_data, bool_dec, moving_dec, ADD_OP, SUB_OP

    front = 0
    back = 0
    operand = 0

    if SYNTAX_ERROR == 0:
        term()
        front = term_data

        # 에러가 나지 않았거나 +, - 가 오면 반복
        while (nextToken == ADD_OP or nextToken == SUB_OP) and SYNTAX_ERROR == 0:
            if nextToken == ADD_OP:
                operand = ADD_OP
            elif nextToken == SUB_OP:
                operand = SUB_OP
            lex()
            term()
```

```
            # 계산
            back = term_data
            if operand == ADD_OP:
                front = front + back
            elif operand == SUB_OP:
                front = front - back

    if SYNTAX_ERROR == 0:
        bool_dec = 1
        moving_dec = front
```

C 코드와 기능적으로 동일합니다.
① aexpr 함수는 term() 함수를 호출하고 다음 토큰이 + 또는 -이거나 에러가 나지 않았다면 다시 term() 함수를 호출합니다.
② term 함수의 결과값은 각각 front와 back에 저장하고 판단한 연산자로 도출된 겨로가 값을 moving_dec에 저장합니다. 또한 이 값이 int형이라는 것을 판별할 판별자의 정보도 업데이트합니다.

(14) term

```
# <term> -> <factor> {(* | /) <factor>
def term():
    global SYNTAX_ERROR, nextToken, demical, term_data, MULT_OP, DIV_OP

    front = 0
    back = 0
    operand = 0

    if SYNTAX_ERROR == 0:
        factor()
        if (len(demical)==0)  :
            error()
        else :
            front = int(demical)

        # 에러가 나지 않았거나 *, / 가 오면 반복
        while (nextToken == MULT_OP or nextToken == DIV_OP) and
SYNTAX_ERROR == 0:
            if nextToken == MULT_OP:
                operand = MULT_OP
            elif nextToken == DIV_OP:
                operand = DIV_OP
```

```
            lex()
            factor()

            # 계산
            if (len(demical) == 0):
                error()
            else:
                back = int(demical)
            if operand == MULT_OP:
                front = front * back
            elif operand == DIV_OP:
                front = front // back

    if SYNTAX_ERROR == 0:
        term_data = front
```

C 코드와 기능적으로 동일합니다.
① term 함수는 factor() 함수를 호출하고 다음 토큰이 * 또는 /이거나 에러가 나지 않았다면 다시 factor() 함수를 호출합니다.
② term 함수의 결과값은 각각 front와 back에 저장하고 판단한 연산자로 도출된 결과 값을 term_data에 저장합니다.

(15) factor

```
# <factor> -> <number>
def factor():
    global SYNTAX_ERROR

    if SYNTAX_ERROR == 0:
        number()
```

C 코드와 기능적으로 동일합니다.
① number 함수를 호출합니다.

(16) number

```
# <number> -> <dec>{<dec>}
def number():
    global dec_dect, demical, SYNTAX_ERROR, notdecimal, demical_index

    dec_dect = 0
    demical = ""
```

```
    if SYNTAX_ERROR == 0:
        dec()
        dec_dect += 1

        while notdecimal == 0 and SYNTAX_ERROR == 0:
            dec()

        notdecimal = 0
        demical_index = 0
```

C 코드와 기능적으로 동일합니다.
① 정수를 저장할 demical 함수를 초기화하고 dec() 함수를 호출합니다.
② 에러가 나지 않았거나 정수가 맞다면 계속해서 dec() 함수를 호출합니다.
③ 이후 notdecimal 과 demical_index를 초기화합니다.

(17) dec

```
# <dec> -> 0|1|2|3|4|5|6|7|8|9
def dec():
    global SYNTAX_ERROR, nextToken, lexeme, demical, demical_index, dec_dect, bexpr_error, notdecimal

    if SYNTAX_ERROR == 0:
        if nextToken == INT_LIT:
            demical = demical + lexeme[0]
            demical_index += 1
            lex()
        elif dec_dect == 0:
            error()
            bexpr_error = -1
        else:
            notdecimal = -1
```

C 코드와 기능적으로 동일합니다.
① dec 함수는 demical 배열에 차례대로 값을 저장합니다.
② 만약 첫 번째 dec함수일 때 정수가 아니라면 에러가 납니다. bexpr에 관련하여 오류인 것일 수도 있어 bexpr_error의 값도 바꿔줍니다.
③ 두 번째 이후의 dec함수가 정수가 아니라면 이후에 판별을 합니다.

(18) var

```
# <var> -> x | y | z
def var():
```

```
    global SYNTAX_ERROR, nextToken, var_detect, VAR_X, VAR_Y, VAR_Z

    if SYNTAX_ERROR == 0:
        if nextToken in [VAR_X, VAR_Y, VAR_Z]:
            if nextToken == VAR_X:
                var_detect = VAR_X
            elif nextToken == VAR_Y:
                var_detect = VAR_Y
            else:
                var_detect = VAR_Z
            lex()
        else:
            error()
```

C 코드와 기능적으로 동일
① var() 함수는 x, y, z인지를 판별하여 var_detect의 값을 업데이트합니다.
② x, y, z가 아니라면 에러를 호출합니다.

(19) error

```
def error():
    global SYNTAX_ERROR
    SYNTAX_ERROR = -1
```

C 코드와 기능적으로 동일합니다.
① SYNTAX_ERROR의 값을 -1로 바꾸어줍니다.