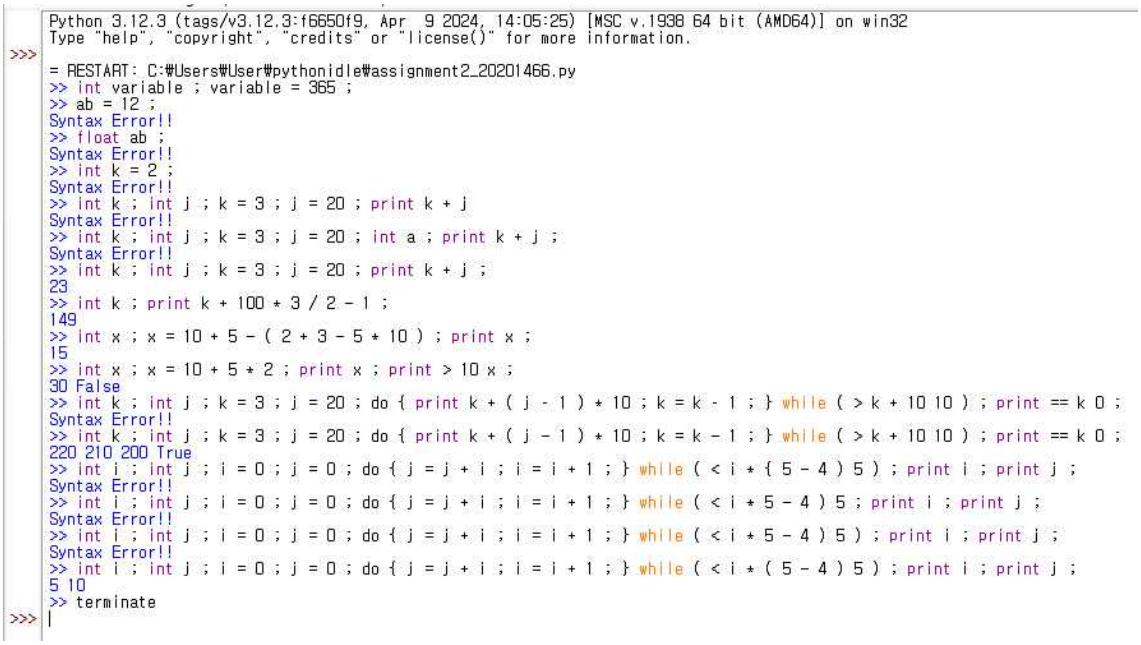
프로그래밍 언어
구현 과제 #2
(PYTHON)

학번	20201466
이름	박사성

목차

1. 실행 결과

파이썬 idle로 실행한 결과입니다.



```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25) [MSC v.1938 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:#Users#User#pythonidle#assignment2_20201466.py
>> int variable ; variable = 365 ;
>> ab = 12 ;
Syntax Error!!
>> float ab ;
Syntax Error!!
>> int k = 2 ;
Syntax Error!!
>> int k ; int j ; k = 3 ; j = 20 ; print k + j
Syntax Error!!
>> int k ; int j ; k = 3 ; j = 20 ; int a ; print k + j ;
Syntax Error!!
>> int k ; int j ; k = 3 ; j = 20 ; print k + j ;
23
>> int k ; print k + 100 * 3 / 2 - 1 ;
149
>> int x ; x = 10 + 5 - ( 2 + 3 - 5 * 10 ) ; print x ;
15
>> int x ; x = 10 + 5 * 2 ; print x ; print > 10 x ;
30 False
>> int k ; int j ; k = 3 ; j = 20 ; do { print k + ( j - 1 ) * 10 ; k = k - 1 ; } while ( > k + 10 10 ) ; print == k 0 ;
Syntax Error!!
>> int k ; int j ; k = 3 ; j = 20 ; do { print k + ( j - 1 ) * 10 ; k = k - 1 ; } while ( > k + 10 10 ) ; print == k 0 ;
220 210 200 True
>> int i ; int j ; i = 0 ; j = 0 ; do { j = j + i ; i = i + 1 ; } while ( < i * { 5 - 4 ) 5 ) ; print i ; print j ;
Syntax Error!!
>> int i ; int j ; i = 0 ; j = 0 ; do { j = j + i ; i = i + 1 ; } while ( < i * 5 - 4 ) 5 ; print i ; print j ;
Syntax Error!!
>> int i ; int j ; i = 0 ; j = 0 ; do { j = j + i ; i = i + 1 ; } while ( < i * 5 - 4 ) 5 ) ; print i ; print j ;
Syntax Error!!
>> int i ; int j ; i = 0 ; j = 0 ; do { j = j + i ; i = i + 1 ; } while ( < i * ( 5 - 4 ) 5 ) ; print i ; print j ;
5 10
>> terminate
>>>
```

2. 코드 설명

(1) 전역변수 설정

```
# 문자 분류 상수
TokenInfo = -1
LETTER = 1       # 문자로 시작
DIGIT = 2        # 숫자로 시작
UNKNOWN = 99     # 알 수 없음 (연산자로 시작)

# 토큰 상수
NULL = 0        #
INT_LIT = 10     #
IDENT = 11       #
OPERATION = 12    #
MIX = 13        #

# 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    # ;
LEFT_BRACE = 33    # {
RIGHT_BRACE = 34   # }
LEFT_PARENTHESES = 35  # (
RIGHT_PARENTHESES = 36  # )

# 문자 상수
SMALL_LETTER = 37
CAPITAL_LETTER = 38


# 전역 변수
charClass = 0
input_str = ""
input_str_index = 0
lexeme = ""
nextChar = ""
token = 0
nextToken = 0
```

```
# 에러 판별자
SYNTAX_ERROR = 0
declaration_error = 0

# 숫자열, 문자열
lower_string = ""
demical = ""

# 계산 결과 저장 배열
print_final = []
printdata = 0

# 임시 저장 변수
# 변수를 리스트로 저장 -> 검색 가능
# var_list = []
var_dict = {}

# 변수 및 연산자 확인 변수
cmp_detect = 0

# 전달하는 값
moving_bool = False
moving_dec = 0
moving_term = 0
```

① 프로그램에서 사용할 변수들을 선언해 주었습니다.

(2) 메인 부분

```
if __name__ == "__main__":
    while True:
        # 초기화
        SYNTAX_ERROR = 0
        input_str = ''
        input_str_index = 0
        print_final = []
        var_dict = {}
        moving_bool = False
        moving_dec = 0
        moving_term = 0
        cmp_detect = 0


        # 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
    if SYNTAX_ERROR == 0:
      if len(print_final) > 0:
        for i in print_final:
          print(i, end=" ")
        print()
    else:
      print("Syntax Error!!")
```

① 초기화 할 변수들을 초기화 해줍니다.
② input_str에 문자열을 입력받습니다.
③ program()을 통해 파싱을 시작합니다.
④ 파싱이 끝난 이후에 오류 발생 여부를 확인하여 알맞은 출력 값을 출력합니다.

(3) lookup_op_1 함수

```
# 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,  LEFT_BRACE,  RIGHT_BRACE,
LEFT_PARENTHESES, RIGHT_PARENTHESES

   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
  elif ch == '{':
    addChar()
    nextToken = LEFT_BRACE
  elif ch == '}':
    addChar()
    nextToken = RIGHT_BRACE
  elif ch == '(':
    addChar()
    nextToken = LEFT_PARENTHESES
  elif ch == ')':
    addChar()
    nextToken = RIGHT_PARENTHESES
  else:
    addChar()
    nextToken = OPERATION
  return nextToken
```

① 연산자 문자열 중 첫 연산자를 판별하는 함수입니다.
② +, -, *, /, = , !, <. >, ;, {, }, (,)을 판별합니다.

(4) lookup_op_2 함수

```
# 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
```

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

(5) lookup_char 함수

```
# 문자가 소문자인지
def lookup_char(ch) :
  global nextToken

  if ch.islower() and (nextToken==SMALL_LETTER or len(lexeme)==0):
    addChar()
    nextToken = SMALL_LETTER
  else:
    addChar()
    nextToken = IDENT

  return nextToken
```

① 문자열이 소문자로만 이루어지는지 판별하는 함수입니다.

(6) lookup_digit 함수

```
# 숫자인지 아닌지
def lookup_digit(ch):
  global nextToken

  if ch.isdigit() and (nextToken==INT_LIT or len(lexeme)==0):
    addChar()
    nextToken = INT_LIT
  else:
    addChar()
    nextToken = MIX

  return nextToken
```

① 문자열이 숫자로만 이루어졌는지 판별합니다.

(7) addChar, getChar, getNonBlank 함수

```
# 임시버퍼에 토큰 저장
def addChar():
   global 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():
   global nextChar
   while nextChar.isspace():
     getChar()
```

① addChar 함수는 판별할 토큰을 배열에 저장해주는 함수입니다.
② getChar 함수는 저장할 문자가 문자인지, 연산자인지, 숫자인지, 아니면 문자열이 끝나는지를 판별해주는 함수입니다.
③ getNoneBlank 함수는 문자열에서 빈칸 이후의 문자열을 받을 수 있게하는 문자열입니다.

(8) lex 함수

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

   # 문자는 다르지
```

```
  if charClass == LETTER:
    TokenInfo = LETTER
    # lookup_char(nextChar)
    # getChar()
    while nextChar != '\0' and not nextChar.isspace():
      lookup_char(nextChar)
      getChar()

  # 숫자도 같애
  elif charClass == DIGIT:
    TokenInfo = DIGIT
    # lookup_digit(nextChar)
    # getChar()
    while nextChar != '\0' and not nextChar.isspace():
      lookup_digit(nextChar)
      getChar()

  # 연산자는 같고
  elif charClass == UNKNOWN:
    TokenInfo = UNKNOWN
    lookup_op_1(nextChar)
    getChar()

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

  elif charClass == NULL:
    TokenInfo = NULL
    nextToken = NULL
    lexeme = "NULL"

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

① lex 함수는 현재 토큰의 정보를 판별해주는 함수입니다.
② 첫 시작이 문자든 숫자든 연산자든 상관없이 공백이 나올 대 까지의 문자열을 판별합니다. 판별하기 위해 (lookup_op_1, lookup_op_2, lookup_char, lookup_digit) 함수를 사용합니다.

(9) program 함수

```
# <program> -> {<declaration>} {<statemnet>}
# 여기서 declaration에서 statement로 이동할 판별자를 만들어줘야 할 듯
def program():
   global declaration_error, nextToken, SYNTAX_ERROR
   declaration_error = 0
   SYNTAX_ERROR = 0

   lex()

   while nextToken!=NULL and SYNTAX_ERROR==0 and declaration_error==0:
     # program_index = input_str_index
     declaration()

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

① <program> -> {<declartion>} {<statement>} 규칙을 행하는 함수입니다.
② declartion 부분이 시작한다면 declartion 함수를 호출하고 반복하다가 statememt 부분이 시작된다면 statement함수를 호출하고 반복합니다.

(10) declaration 함수

```
# <declaration> -> <type> <var> ;
# 변수 리스트에 추가
def declaration():
   global declaration_error, var_dict, SYNTAX_ERROR
   # 여기서는 아직 에러 판단을 하지 않아
   # 왜? declaration이 하나도 안올 수 있으니까
   declaration_error = 0

   type()

   if declaration_error==0:
     var()

     # 변수 등록
     var_dict[lower_string] = 0
     # print(var_dict)

   if declaration_error==0 and SYNTAX_ERROR==0:
     if nextToken == SEMI_COLON:
       lex()
     else :
       error()
```

① <declaration> -> <type> <var> ; 규칙을 행하는 함수입니다.
② type 함수를 호출한 후 오류가 발생하지 않았다면 var함수를 호출하고 그 결과값을 0으로 초기화하여 var_dict에 저장합니다.
③ 이후 ; 가오는지 확인하여 아니라면 에러 함수를 호출합니다.

(11) statement() 함수

```
# <statement> → <var> = <aexpr> ; | print <bexpr> ; | print <aexpr> ; | do ‘ {’ {<statement>} ‘ } ’ while ( <bexpr> ) ;
# 문자를 하나씩 받는데 print하고 do를 어떻게 구분할 것인가
# 그래서 <var> = <aexpr> ; 이거를 맨 처음에 둔거같은데?
def statement():
    global  nextToken,  ASSIGN OP,  SYNTAX ERROR,  input str index,  printdata, moving_bool, moving_dec
   # <var> = <aexpr> ;
   if SYNTAX_ERROR==0 :
     if lexeme=="print" :
       lex()

       if nextToken==LESS_THAN or nextToken==MORE_THAN or nextToken==LESS or nextToken==MORE or nextToken==EQUAL or nextToken==NOT_EQUAL :
         bexpr()
         printdata = moving_bool
       else :
         aexpr()
         printdata = moving_dec

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

       if SYNTAX_ERROR==0:
         print_final.append(printdata)


     elif lexeme=="do" :
       while SYNTAX_ERROR==0:
         loop_dir = False
         expr_index = input_str_index

         lex()

         if SYNTAX_ERROR==0:
           if nextToken==LEFT_BRACE:
             lex()
           else :
```

```
            error()

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

        if SYNTAX_ERROR==0:
          if nextToken==RIGHT_BRACE :
            lex()
          else :
            error()

        if SYNTAX_ERROR==0:
          if lexeme=="while" :
            lex()
          else :
            error()

        if SYNTAX_ERROR==0:
          if nextToken==LEFT_PARENTHESES:
            lex()
          else :
            error()

        if SYNTAX_ERROR==0:
          bexpr()
          loop_dir = moving_bool

        if SYNTAX_ERROR==0:
          if nextToken==RIGHT_PARENTHESES:
            lex()
          else :
            error()

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

        if SYNTAX_ERROR==0:
          if loop_dir == True:
            input_str_index = expr_index
          else :
            break

    else :
      variation = ""
```

```
    var()

    # 변수가 선언되었었는지 확인
    if SYNTAX_ERROR == 0:
      if lower_string in var_dict:
        variation = lower_string
      else:
        error()

    if SYNTAX_ERROR==0:
      if nextToken==ASSIGN_OP:
        lex()
      else :
        error()

    if SYNTAX_ERROR==0:
      aexpr()

    if SYNTAX_ERROR==0:
      var_dict[variation] = moving_dec

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

① <statement> -> <var> = <aexpr> | print <bexpr> | print <aexpr> | do '{' {<statement>} '}' while (<bexpr>) ; 규칙을 행하는 함수입니다.
② 다음 토큰이 print라면 다음에 오는 토큰의 정보에 따라 aexpr 이나 bexpr을 호출합니다. 또한 결과값을 print_final 리스트에 저장합니다.
③ 다음 토큰이 do 라면 차례로 '{'을 판별, statement 함수 호출 반복, '}'을 판별, while 판별 '('판별 bexpr함수 호출, ')' 판별, ;을 판별합니다.
④ 만약 bexpr의 결과값이 참이라면 do를 판별한후 저장했었던 input_str_index를 활용해 다시 반복합니다.
⑤ 다음 토큰이 다른 것이라면 var함수를 호출한 후, = 판별, 그 후 aexpr 함수를 호출합니다.

(12) bexpr() 함수

```
# <bexpr> → <relop> <aexpr> <aexpr>
def bexpr():
   global SYNTAX_ERROR, cmp_detect, moving_bool, moving_dec
   moving_bool = False

   relop()

   if SYNTAX_ERROR==0:
     aexpr()
     if SYNTAX_ERROR==0:
       front = moving_dec

   if SYNTAX_ERROR==0:
     aexpr()
     if SYNTAX_ERROR==0:
       back = moving_dec

   if SYNTAX_ERROR==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)

     moving_bool = bool_bexpr
```

① <bexpr> -> <relop> <aexpr> <aexpr> 규칙을 수행하는 함수입니다.
② relop 함수를 호출합니다.
③ 에러가 나지 않았다면 차례로 aexpr 함수를 호출하고 각각 front와 back에 결과값을 저장해줍니다.
④ 다음 relop의 결과 값을 확인하여 bexpr의 결과값을 moving_bool에 저장해줍니다.

(13) relop 함수

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

   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()

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

① <relop> -> == | != | < | > | <= | >= 규칙을 수행하는 함수입니다.
② 토큰의 정보를 활용하여 cmp_detect에 저장해줍니다.

(14) aexpr 함수

```
# <aexpr> → <term> {( + | - | * | / ) <term>}
def aexpr():
    global SYNTAX_ERROR, nextToken, moving_term, moving_dec, ADD_OP, SUB_OP,
MULT_OP, DIV_OP

   front = 0
   back = 0
   operand = 0
   moving_dec = 0

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

     while ((nextToken==ADD_OP or nextToken==SUB_OP or nextToken==MULT_OP or
nextToken==DIV_OP) and  SYNTAX_ERROR==0):
       if nextToken == ADD_OP:
         operand = ADD_OP
```

```
      elif nextToken == SUB_OP:
        operand = SUB_OP
      elif nextToken == MULT_OP:
        operand = MULT_OP
      elif nextToken == DIV_OP:
        operand = DIV_OP
      lex()
      term()

      if SYNTAX_ERROR==0:
        # 계산
        back = moving_term
        if operand == ADD_OP:
          front = front + back
        elif operand == SUB_OP:
          front = front - back
        elif operand == MULT_OP:
          front = front * back
        elif operand == DIV_OP:
          front = int(front / back)

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

① <aexpr> -> <term> { (+|-|*|/) <term> } 규칙을 수행하는 함수입니다.
② term함수를 호출한후 조건에 맞다면 operand에 정보를 저장하고 lex 함수와 term 함수를 호출합니다.
③ term의 결과값을 각각 front와 back에 저장하고 aexpr의 결과값은 moving_dec에 저장합니다.

(15) term 함수

```
# <term> → <number> | <var> | ( <aexpr> )
def term():
  global SYNTAX_ERROR, var_dict, moving_term
  # number
  if TokenInfo==DIGIT :
    number()
    if SYNTAX_ERROR==0:
      moving_term = demical

  # var
  elif TokenInfo==LETTER :
    var()
    if SYNTAX_ERROR==0:
      # print(lower_string)
      # print(var_dict)
```

```
      if lower_string in var_dict:
        moving_term = var_dict[lower_string]
      else :
        error()

  # ( aexpr )
  elif TokenInfo==UNKNOWN :
    if nextToken==LEFT_PARENTHESES:
      lex()
    else :
      error()

    if SYNTAX_ERROR==0:
      aexpr()
      if SYNTAX_ERROR==0:
        moving_term = moving_dec

    if SYNTAX_ERROR==0:
      if nextToken==RIGHT_PARENTHESES:
        lex()
      else :
        error()
  # error
  else :
    error()
```

① <term> -> <number> | <var> | (<aexpr>) 규칙을 수행하는 함수입니다.
② 토큰의 정보에 따라 var, number 함수를 호출하거나 (을 판별하고 aexpr 함수를 호출하고)을 판별합니다.
③ 각 함수 수행 결과값을 movint_term에 저장합니다.

(16) type 함수

```
# <type> → int
# declartion int가 아니면 바뀐다.
# 문자를 하나씩만 받아서 함수를 호출 해야해
# <var> 로 받으면 안되나?
def type():
  global declaration_error, lexeme

  # int 인지 아닌지 확인
  if lexeme == "int":
    lex()
  else :
    declaration_error = -1
```

① 다음 토큰이 'int'인지 확입합니다.

(17) number 함수

```
# <number> → <dec>{<dec>}
# <dec> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
def number():
   global demical, SYNTAX_ERROR
   if SYNTAX_ERROR == 0:
     if nextToken == INT_LIT:
       demical = int(lexeme)
       lex()
     else:
       error()
```

① 토큰이 숫자로만 이루어져있는지 확인하고 숫자로 이루어진 문자열을 정수로 바꿔 decimal에 저장합니다.

(18) var 함수

```
# <var> → <alphabet>{<alphabet>}
# <alphabet> → a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q
| r | s | t | u | v | w | x | y | z
def var():
   global lower_string, SYNTAX_ERROR

   if SYNTAX_ERROR == 0:
     if nextToken == SMALL_LETTER :
       lower_string = lexeme
       lex()
     else :
       error()
```

① 토큰이 소문자로만 이루어졌는지를 판단하고 lower_string에 토큰을 저장합니다.

(19) error 함수

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

① SYNTAX_ERROR 값을 -1로 바꿉니다.