프로그래밍 언어 과제2

20193010 심상현

선택사항 구현여부 O

선택사항이었던 EBNF의 factor에서 [-]를 Python, C++, java에서 모두 구현 완료했습니다.

함수 목차

구현 방법은 C++코드로 모두 통일하여 설명하였습니다.

int main()

실제 코드가 돌아가는 main()에서는 반복문을 계속 돌려주는 방식을 사용하였습니다.
먼저 queue에 들어가 있는 값을 초기화 해주고, 키보드에서 입력받은 값을 str에 저장합니다.
그리고 `tokenParse()`에 str의 값들을 토큰 단위로 분석하여 저장하여 전역변수로 설정되어 있는 queue에 넣어줍니다. (정의되어 있는 값 : `queue<string> Q;`)
다음 계산을 위해서 `expr()`를 호출하고 나온 최종 값을 ans에 저장하여 출력해줍니다. 만약 모든 계산이 종료된 시점에서 queue안에 데이터가 남아있다면 syntax error를 찾은 것이기 때문에 `error()`를 호출합니다. *(예시 : 입력 데이터가 "(78*2))"인 경우 ')'가 queue에 남아있음)* 계산이 종료되었다면 최종 값인 ans를 출력하고 다시 입력을 받습니다.

하지만 Java의 경우 조금 다른데, C++나 Python의 경우 `cout` 혹은 `print()`를 사용하면 해당 값의 형태에 따라 실수 혹은 정수로 자동 변환이 이루어지지만, Java는 타입체킹을 강하게 하는 언어여서 해당 값에 따라 `Math.ceil(ans) - ans`에서 반환 값이 0이라면 해당 값이 실수라고 인지해서 출력을 실수형으로, 아니라면 정수형으로 출력해줍니다.

```
int main() {
  while(1) {
    int ans = 0;
    string str;
    queueClear();
    cout << ">> ";
    getline(cin, str);
    tokenParse(str);
    ans = expr();
    //if input is left
    if(!Q.empty()) error();
    cout << ans << '\n';
  }
```

```
if(Math.ceil(ans) - ans != 0) //if data is floating point
  System.out.println(ans);
else //if data is integer
  System.out.println((int)ans);
```

void error()

프로그램을 돌아가던 중에 syntax error가 발견될 경우, `error()`를 호출합니다.
해당 함수는 "syntax error!!"를 출력하고, 현재 돌아가고 있는 프로세스를 종료합니다.

```
void error() {
  cout << "syntax error!!\n";
  queueClear();
  //process end
  exit(0);
}
```

void tokenParse(string)

string형태로 받은 input값들을 토큰 단위로 잘라서 queue에 넣어줍니다.
입력 string의 값들을 하나씩 분석하는데, 해당 값이 띄어쓰기(' ')이라면 건너뛰기, 숫자라면 해당 숫자만큼을 저장하는 함수인 `DIGIT()`으로 넘겨줍니다. 그리고 계산 문자(*, /, +, -, (,))의 경우에는 해당 값을 바로 queue에 넣어줍니다. 그 이외의 상황은 모두 error을 의미함으로 `error()`를 호출합니다.

```
void tokenParse(string str) {
  for(int i=0; i<str.length(); i++) {
    if(str[i] == ' ') continue;
    else if(str[i] >= '0' && str[i] <= '9') {
      i += DIGIT(str, i);
    }
    else if(str[i] == '*' || str[i] == '/' || str[i] == '+' ||
        str[i] == '-' || str[i] == '(' || str[i] == ')') {
      string temp = "";
      temp += str[i];
      Q.push(temp);
    }
    else {
      error();
    }
  }
}
```

int DIGIT(string, int)

input으로 받은 string을 문자 1개씩 분석하다가 숫자가 나오는 경우, 그 숫자를 인식하기 위해 만든 함수입니다.
반복문으로 tokenParse()에서 숫자가 있다는 index부터 string의 길이까지 확인해주는데 하나씩 읽으면서, 숫자가 나오면 이전 숫자의 뒤에 붙이는 형식으로 자릿수를 늘리는 것을 구현하였습니다. 띄어쓰기가 나오면 `continue`를 해줍니다. 숫자, 공백 이외의 문자가 들어있거나, string의 끝까지 확인했다면 해당 더 이상 확인할 문자가 없다는 것으로 인지하고 종료합니다.
한번 반복문을 돌아갈때마다 cnt값을 +1해주는데, 이를 이용해서 `DIGIT()`에서 얼만큼의 index를 읽었고, `tokenParse()`에서 중복해서 읽지 않아도 되는 구간을 알려주기 위해서 cnt변수를 추가하였습니다.
반복문에서 추가해준 문자를 queue에 추가해주고 cnt값을 돌려줍니다.

```
int DIGIT(string str, int index) {
  string temp = "";
  int cnt = 0;
  for(int i=index; i<str.length(); i++, cnt++) {
    if(str[i] >= '0' && str[i] <= '9') temp += str[i];
    else if(str[i] == ' ') continue;
    else break;
  }
  Q.push(temp);
  return cnt-1; //i++ is already in tokenParse()
}
```

double expr()

EBNF의 <expr>를 나타내는 함수입니다.
맨처음 data 변수를 설정하고 <term>에 계산된 값으로 초기화 해줍니다.
lex()로 다음 읽은 값이 '+'라면 다음 <term>을 더해주고, '-'라면 <term>을 빼줍니다.
다음 값이 '+'나 '-'라면 위의 메커니즘을 계속 반복해주고 아니라면 반복문으로 종료하고 계산을 완료한 값을 return해줍니다.

이후부터 non-terminal에 해당되는 값들을 계산하고, 반환하기 위해 double형으로 data를 설정하고, return type을 double로 설정하여, 나눗셈이 발생하여 실수형으로 되었을 때, 값에 대해 유연한 대입이 가능합니다.

```
double expr() {
  double data = 0;
  data = term();
```

```
  while(nextValue() == "+" || nextValue() == "-") {
    char oper;
    oper = lex()[0];

    if(oper == '+') data += term();
    else if(oper == '-') data -= term();
  }
  return data;
}
```

double term()

EBNF의 <term>을 나타내는 함수입니다.
data 변수를 설정하고, <factor>로 초기화해줍니다.
이후 반복문에 들어가서 다음 값이 '*'이나 '/'라면 계속 반복해줍니다. `lex()`로 다음으로 할 operation이 무엇인지 oper 변수에 설정하고, 해당 값에 따라 계산을 실시합니다. 반복문이 종료되면, data를 return합니다.

```
double term() {
  double data = 0;
  data = factor();
  while(nextValue() == "*" || nextValue() == "/") {
    char oper;
    oper = lex()[0];
    if(oper == '*') data *= factor();
    else if(oper == '/') data /= factor();
  }
  return data;
}
```

double factor()

EBNF의 <factor>를 나타내는 함수입니다.
여기서 선택사항 구현부분이 들어가 있는데, `nextValue()`를 이용해서 다음 값을 읽어오고, 해당 값이 '-'이라면 `factor()` 마지막 return값을 음/양수 부호를 바꿔서 return합니다.
다음으로 만약 `nextValue()`로 받아온 첫번째 값이 숫자라면 data를 `number()`를 호출하여 숫자로 이루어진 다음 token을 저장합니다.
만약 숫자가 아니라면 괄호가 있는 형태로 인지하고, `nextValue()`의 값이 '('이고, 다음 `expr()`를 호출한 뒤 `nextValue()`이 ')'가 아닌, 이외의 경우 `error()`를 호출합니다.
그 이후 위에서 선택 구현 사항인 oper값을 확인해주고, -라면 -data를 반환, 아니라면 data를 반환합니다.

여기서도 Java의 강한 타입체킹에 대해 약간의 추가적인 코드가 필요합니다.
factor()에서 추가적으로 구현한 [-]의 값을 생각해보면, `nextValue()`의 값이 더 이상 존재하지 않는 경우에 대해서 Java의 경우 `nextValue().charAt(0)`에 대한 연산은 빈 string에 대한 index 0을 접근을 요구하기 때문에, `OutOfBoundException`을 발생시킬수 있습니다. 그래서 해당 조건에 대한 예외를 설정해야합니다. `if(nextValue() == "") error();`를 미리 넣어주어 입력이 "-"만 들어올 경우에 대해 예외 처리를 설정합니다.

```
double factor() {
  double data = 0;
  //if nextValue is '-' (option)
  char oper = '+';
  if(nextValue() == "-") {
    oper = lex()[0];
  }
  //if nextValue is digit
  if(nextValue()[0] >= '0' && nextValue()[0] <= '9') data = number();
  //( <expr> )
  else {
    if(nextValue() == "(") {
      lex();
      data = expr();
      if(nextValue() == ")") {
        lex();
      }
      else error();
    }
    else error();
  }
  if(oper == '-') return -data;
  else return data;
}
```

int number()

lex()를 이용해 queue의 front값을 읽어오고, 해당 값을 int형으로 변환하여 반환합니다.

```
int number() {
  string NUM = lex();
  return stoi(NUM);
}
```

string lex()

만약 queue가 비어있는 경우에는 아무것도 없는 “”를 반환시켜주고, 만약 비어있지 않은 경우에는 queue의 pop한 다음 해당 데이터를 반환합니다.

```
string lex() {
  if(!Q.empty()) {
    string value = Q.front();
    Q.pop();
    return value;
  }
  else return "";
}
```

string nextValue()

queue가 비어있다면 빈 string “”를 반환, 비어있지 않다면 queue의 front값을 반환합니다.

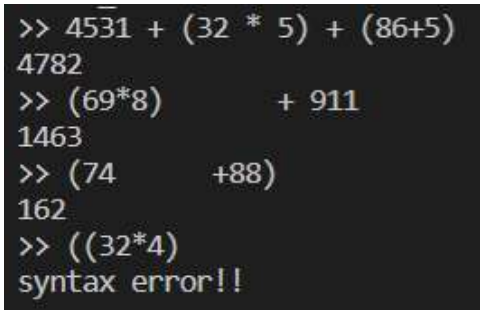
```
string nextValue() {
  if (!Q.empty()) return Q.front();
  else return "";
}
```

void queueClear()

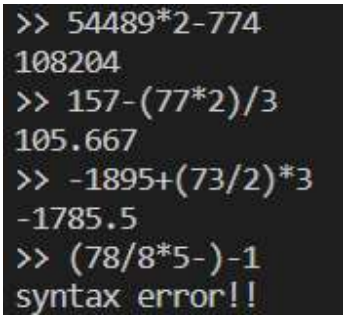
queue의 값이 들어있다면 pop을 계속 해주어서, 결국 queue안에 데이터가 없도록 만드는 함수입니다.

```
void queueClear() {
  while(!Q.empty()) Q.pop();
}
```

실행결과



괄호가 모두 안 닫히는 경우



문자가 오면 안되는 위치에 문자가 들어올 경우