Scanner

Development environment

OS: MacOS Sonoma 14.0

compiler:

Apple clang version 14.0.3 (clang-1403.0.22.14.1)

flex 2.6.4 Apple(flex-34)

complie method: Using the Make file that exist in the directory

(in terminal, type make clean → make → run(ex. ./cminus_cimpl or ./cminus_lex))

Project Specification

implement C-minus Scanner

1. Method 1: C code

   Recognize tokens by DFA

   Scanner는 기본적으로 string → tokenize → Regular Expression → DFA → recognize의 방식을 거친다. 해당 명세에 따라 string을 토큰화해서 DFA에 안정적으로 넘겨주는 것까지를 구현할 것이다.

2. Method 2: Lex(flex)

   Lexical pattern을 Regular expression으로 특정한다.

Point

1. Comments: /* */ (no single line comment) (if comment is opened but not closed it result EOF)

Implement

Method 1

global.h - Token 목록 및 갯수 수정

```
/* MAXRESERVED = the number of reserved words */
#define MAXRESERVED 6

typedef enum
    /* book-keeping tokens */
   {ENDFILE,ERROR,
    /* reserved words */
    //IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,
    IF, ELSE, WHILE, RETURN, INT, VOID,
    /* multicharacter tokens */
    ID,NUM,
    /* special symbols */
    ASSIGN,EQ,NE,LT,LE,GT,GE,PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,LBRACE,RBRACE,LCURLY,RCURLY,SEMI,COMMA
   } TokenType;

extern FILE* source; /* source code text file */
```

MAXRESERVED와 enum을 주어진 명세에 맞게 수정한다.

utils.c - Scanner의 print를 형식에 맞게 맞춰준다.

```
{ case IF:
  case ELSE:
  case WHILE:
  case RETURN:
  case INT:
  case VOID:
    fprintf(listing,
        "reserved word: %s\n",tokenString);
    break;
  case ASSIGN: fprintf(listing,"=\n"); break;
  case LT: fprintf(listing,"<\n"); break;
  case EQ: fprintf(listing,"==\n"); break;
  case NE: fprintf(listing, "!=\n"); break;
  case LE: fprintf(listing, "<=\n"); break;
  case GT: fprintf(listing, ">\n"); break;
  case GE: fprintf(listing, ">=\n"); break;
  case LPAREN: fprintf(listing,"(\n"); break;
  case RPAREN: fprintf(listing,")\n"); break;
  case LBRACE: fprintf(listing, "[\n"); break;
  case RBRACE: fprintf(listing, "]\n"); break;
  case LCURLY: fprintf(listing, "{\n"); break;
  case RCURLY: fprintf(listing, "}\n"); break;
  case COMMA: fprintf(listing, ",\n"); break;
  case SEMI: fprintf(listing,";\n"); break;
  case PLUS: fprintf(listing,"+\n"); break;
  case MINUS: fprintf(listing,"-\n"); break;
  case TIMES: fprintf(listing,"*\n"); break;
  case OVER: fprintf(listing,"/\n"); break;
  case ENDFILE: fprintf(listing,"EOF\n"); break;
  case NUM:
```

Scan.c - 실질적으로 scan을 하는 코드로 해당 파일에서 DFA를 명세에 맞게 구현해주어야 한다.

int getNextChar() = lookahead(1)을 하는 함수

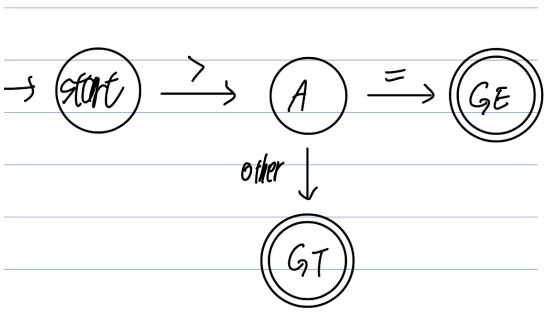
void UngetNextChar() = lookahead로 버퍼에 담은 것을 삭제하는 함수

getToken() = getNextChar을 통해 얻은 character를 token화하는 함수. 여기서 DFA를 알맞게 구현한다.

기본적으로 DFA를 구현할 때 특정 symbol이 바로 final state로 가게 만드는 경우는 다음과 같이 간단하게 구현해줄 수 있다.

```
case '+':
  currentToken = PLUS;
  break;
```

바로 final state로 가기에 currentToken에 해당 symbol에 맞는 token을 할당해주면 된다.



우리가 적절하게 구현해야하는 부분은 각 symbol이 여러개의 의미를 가질 수 있을 경우이다.

예를 들어, getToken을 했을 때 < 라는 char을 얻었다면 이는 < (GT) 이거나 <= (GE) 일 수 있다.

이런 예외 처리를 안해주면 DFA가 아닌 NFA가 형성되고 scannr는 제대로 작동할 수 없을 것이다.

다음과 같은 DFA를 그릴 수 있다. 이를 코드로 구현한다면,

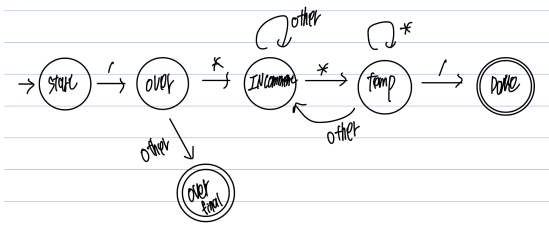
```
else if (c == '>'){
 state = INGRE;
}
```

getNextChar을 통해 얻은 c가 만약 > 과 같다면 state를 INGRE(위 그림에서의 A state)로 옮겨준다.

```
case INGRE:
 state = DONE;
 if(c == '='){
   currentToken = GE;
 }
 else{
   currentToken = GT;
 }
 break;
```

switch문이기에 코드는 해당하는 조건이 나올때까지 쭉 내려올 것이다.>

이후 break를 만나고 다시 getNextChar을 통해 다음 char을 가져온다. 현재 state는 INGRE이고 c에는 >다음에 나온 char가 담겨있다. INGRE case에서 c에 따라 currentToken을 결정한다. 둘 다 final state이므로 state는 해당 부분에서 DONE으로 바꿔준다.



**Commnet Handling

construct comment DFA

처음 /*을 인식하는 것은 비교적 간단하지만 마지막 */을 인식할 때에는 이들이 연속적으로 나오지 않는 경우가 있다는 것을 잘 파악해서 DFA를 설계하고 코드를 써야한다.

```
160       case INCOMMENT:
161         save = FALSE;
162         if (c == EOF)
163         { state = DONE;
164           currentToken = ENDFILE;
165         }
166         else if (c == '*') state = INCOMMENTOVER;
167         break;
168       case INCOMMENTOVER:
169         save = FALSE;
170       if(c == EOF){
171         state = DONE;
172         currentToken = ENDFILE;
173       }
174       else if(c == '/'){
175           tokenStringIndex = 0;
176           state = START;
177       }
178       else{
179         state = INCOMMENT;
180       }
181       break;
```

그래서 옆의 코드와 같이 / 즉 OVER state에서 다음 input으로 *을 만나면 INCOMMENT state로 온다.

그 상태에서 *을 만나면 INCOMMENTOVER state(그림상 temp)로 넘어가는데 이때 바로 다음 input에 /가 오지 않았을 때의 transition을 잘 처리해준다.

연속적으로 */가 나왔을 경우에만 INCOMMENTOVER에서 DONE으로 넘어갈 수 있도록 한다.

Method2

lex는 Regular expression으로 표현하므로 token으로 분류해야하는 것들을 모두 rule section에 넣어주면 된다.

comment 외의 다른 token의 경우 input을 받으면 return을 그대로 해주면 된다.

```
16  digit       [0-9]
17  number      {digit}+
18  letter      [a-zA-Z]
19  identifier  {letter}({letter}|{digit})*
20  newline     \n
21  whitespace  [ \t]+
```

다만, identifier는 string이 문자로 시작한다면 숫자가 나와도 이를 num이 아닌 identifier로 인식해야하기때문에 다음과 같이 rule section의 RE를 수정해준다.

ex.Leeseungsu0051 → identifier, 0051 is not num

```
54  "/*"            { char first;
55                    char second;
56                   while(1)
57                   { first = input();
58                     if (first == '\0') break;
59                     if (first == '\n') lineno++;
60                     if(second == '*' && first == '/') break;
61                     second = first;
62                   };
63                 }
64  .              {return ERROR;}
```

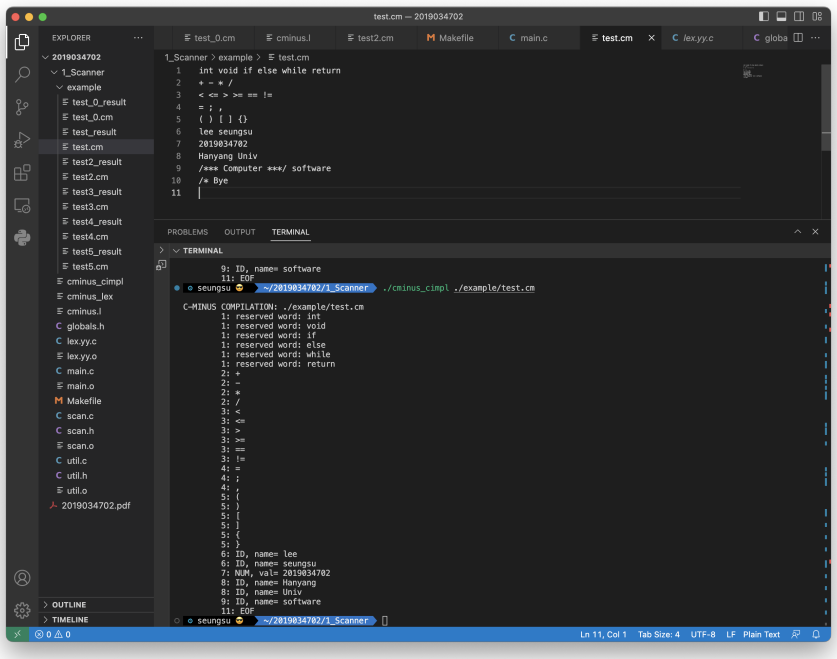
comment 처리의 경우 comment의 */을 만났을 때 return 할 수 있도록 함수의 rule을 만들어주어야 한다.

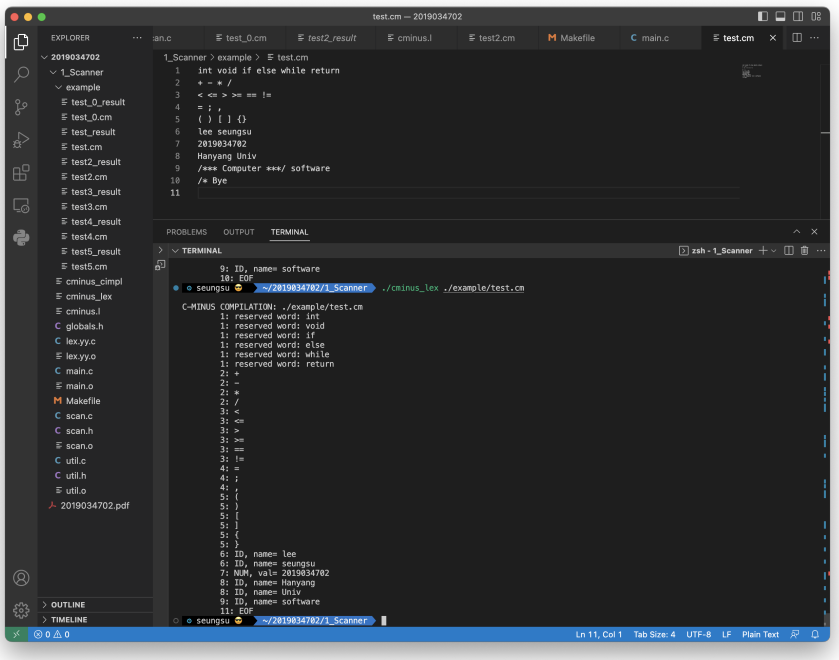
escape을 위해 first와 second 변수를 설정해주고, first와 second이 차례로 *과 / 일 때 break 할 수 있도록 한다.

다만, 처음에 EOF를 lex에서도 EOF로 표현되는 줄 알고 input이 EOF일 때 break하도록 코드를 짰는데, terminate되지 않은 comment를 test할 때 무한루프를 돌며 프로그램이 종료되지 않는 것을 알았다. 여러 글을 찾아보니 FLEX에서 2016년도 버전부터 End of File을 EOF가 아닌 0으로 return한다는 것을 알았고 이를 처리해주니 정상적으로 test가 되었다.

추가로, 명세를 다시 천천히 읽어보니 이에 대한 설명도 해주셨었다. 앞으로 명세를 더 신중하게 읽어야겠다고 다짐했다.

Result





추가적으로 Symbol들을 모두 테스트해보았다. cimple과 lex 모두 정상적으로 의도한 scanner의 역할을 하고 있음을 확인할 수 있었다.

terminate되지 않은 comment나 comment의 종료인 */부분에서 *다음 바로 /가 와야 정상적으로 terminate됨도 알 수 있다.