나 정규 표현 integer로는 일곱 개만이 인식되기 때문에 선택 규칙 ①에 의해 두 번째 정규 표현에 매칭되어 integers가 인식되며 그에 따른 액션 코드가 실행된다. 그러나 입력이 integer인 경우는 정규 표현 integer나 [a-z]+로 인식되는 문자의 개수는 모두 일곱 개로 동일하다. 이러한 경우는 규칙 ②에 의해 먼저 기술된 정규 표현인 integer에 의해 인식되어 "Keyword integer"란 메시지를 출력하게 된다.

렉스의 입력을 작성하고 실행해 보는 통합적인 예로 다음을 생각해 보자. 여기서, 렉스의 입력 파일은 test.l이고 입력 스트림에 해당하는 파일 이름은 test.dat이다.

■ 렉스 파일: test.l

```
%{
#include <stdio.h>
#include <stdlib.h>
enum tnumber { TEOF, TIDEN, TNUM, TASSIGN, TADD, TSEMI, TDOT,
                TBEGIN, TEND, TERROR};
%}
letter    [a-zA-Z_]
digit     [0-9]
%%
begin                          return(TBEGIN);
end                            return(TEND);
{letter}({letter}|{digit})*    return(TIDEN);
":="                           return(TASSIGN);
"+"                            return(TADD);
{digit}+                       return(TNUM);
";"                            return(TSEMI);
\.                             return(TDOT);
[ \t\n]                        ;
.                              return(TERROR);
%%
void main()
{ enum tnumber tn; /* token number */
  printf(" Start of Lex\n");
  while ((tn=yylex()) != TEOF) {
    switch (tn) {
        case TBEGIN : printf("Begin\n"); break;
        case TEND   : printf("End\n");    break;
```

```
            case TIDEN   : printf("Identifier: %s\n", yytext); break;
            case TASSIGN : printf("Assignment_op\n"); break;
            case TADD    : printf("Add_op\n"); break;
            case TNUM    : printf("Number: %d\n", atoi(yytext)); break;
            case TSEMI   : printf("Semicolon\n"); break;
            case TDOT    : printf("Dot\n"); break;
            case TERROR  : printf("Error: %c\n", yytext[0]); break;
        }
    }
}
int yywrap()
{ printf(" End of Lex\n");
  return 1;
}
```

■ 데이터 파일 : test.dat

```
begin
num := 0;
num := num + 526;
end.
```

test.l을 렉스의 입력으로 하여 스캐너를 작성한 후, 생성된 스캐너에 test.dat을 입력으로 하여 실행된 결과는 다음과 같다.

```
$ lex test.l
$ cc lex.yy.c -o test -ll
$ test < test.dat
```

■ 실행 결과 :

```
Start of Lex
Begin
Identifier: num
Assignment_op
Number: 0
Semicolon
Identifier: num
```

```
Assignment_op
Identifier: num
Add_op
Number: 526
Semicolon
End
Dot
End of Lex
```

다음은 Mini C의 어휘 분석기를 생성하기 위하여 작성된 렉스에 대한 입력이다.

```
%{
   /*  lex source for Mini C */
%}
%%
"const"              return(tconst);
"else"               return(telse);
"if"                 return(tif);
"int"                return(tint);
"return"             return(treturn);
"void"               return(tvoid);
"while"              return(twhile);
"=="                 return(tequal);
"!="                 return(tnotequ);
"<="                 return(tlesse);
">="                 return(tgreate);
"&&"                 return(tand);
"||"                 return(tor);
"++"                 return(tinc);
"--"                 return(tdec);
"+="                 return(taddAssign);
"-="                 return(tsubAssign);
"*="                 return(tmulAssign);
"/="                 return(tdivAssign);
"%="                 return(tmodAssign);
[A-Za-z_][A-Za-z0-9_]*          return(tident);
```

```
    [1-9][0-9]*|0([0-7]+|(x|X)[0-9A-Fa-f]*)?      return(tnumber);
    "/*"([^*]|\*+[^*/])*\**"*/"                   ;
    "//".*                                        ;
    [ \t\n]                                       ;
.                       return(yytext[0]);
%%
int yywrap()
{
   return 1;
}
```

여기서 tconst, telse 등은 토큰 번호로 어떤 고유한 값으로 정의된 정수들이다. 위에서 작성된 렉스의 입력은 14장에서 설명하는 Mini C에 대한 YACC의 입력과 함께 Mini C 컴파일러의 전단부를 구성할 수 있다.

렉스의 출력인 lex.yy.c 프로그램을 렉스 소스(*.l)와 함께 비교/분석하면 렉스 소스를 구성하고 있는 각 부분을 더 잘 이해할 수 있을 것이다. 본문에 있는 렉스 소스를 실제로 실행시켜보고 렉스 소스와 비교해 보기 바란다.