Lex Report

B711134 이승환

April 2021

1 Lex의 동작방식

Lex의 동작방식 Lex란? 일정한 구조의 패턴을 인식하여 토큰이라는 단위로 입력을 나눠 어휘 패턴을 인식하는 프로그램 좀더 자세히 보자면 어휘분석기를 자동으로 생성해주는 어휘분석기 생성기라고 보면 된다.
Lex는 사용자가 정의한 토큰의 정규표현식과 수행C코드를 받아 어휘분석기를 출력하고 이후 이 분석기는 입력(input)을 받아 해당하는 토큰을 찾았을 때 첨부된 수행코드를 수행하여 구분 및 output value를 출력한다.
Lex의 구조는 정의절, 규칙절, 서브루틴절 3가지로 나뉜다.

1. 정의절은 말 그대로 정의에 관한 파트이다. 출력을 위한 C프로그램 헤더파일 및 특정한 정규표현식선언 lex프로그램에서 카운트 및 C프로그래밍에 필요한 변수등을 여기에 모두 선언하게 된다. 정의절 선언은

```
%{
    /*정의절 선언내용 */
%}
```

   위와 같이 선언된다.

2. 규칙절은 원하는 패턴(토큰)을 설정하여 그 패턴에 해당하는 input값이 나타났을 때, 원하는 동작(연산)의 구성으로 되어있는 파트이다. 원하는 문자열의 패턴을 모두 설정할 수 있으며 이에 맞는 input값이 인식될 시 수행코드의 연산을 취하게된다. 또한 한번 인식된 input은 이후 다른 패턴에 의해 인식될 수 없게 된다.
   규칙절 파트는 정규표현과 연산 파트와 쌍으로 구성되는데 이때 정의절과의 구분은 구분자%%로 나뉘어진다.

```
%%
정규표현식1        {연산식(수행코드)1};
정규표현식2        {연산식(수행코드)2};
정규표현식3        {연산식(수행코드)3};
```

   이때 인식되는 토큰을 갖기 위해서는 정규표현(Regular Expression)을 활용하여 패턴을 설정해야되는데 이때 쓰는 정규표현식은 특정 메타기호를 이용하여 패턴을 나타낼 수 있다. 다만 모든 메타기호에 대해 여기서는 기술하지 않겠다.

정규표현식 인식에도 일련의 규칙에 따라 선행인식이 달라진다.

1. 먼저 작성된 규칙이 먼저 선택된다.
2. 해당되는 규칙이 없으면 Echo한다. 즉, 그대로 출력한다.
3. 해당되는 규착이 여러개면 길이가 긴것을 우선적으로 인식.

위의 규칙에 따르면 더 명시적인 규칙이 상위에 있어야함을 알 수 있다. 이 부분도 이후 기술할 문제에서 자세히 다룬다. 또한 규칙절에는 시작상태를 이용하여 특수한 규칙을 설정 할 수 있는데 이는 특정한 상황에서만 토큰을 인식하여(ex. 123이 먼저 인식되었을 때 456인식) 사용자로 하여금 더 자유로운 확장성을 갖게한다.
시작 상태 설정으로

```
/* 정의절 */
%}
%x 시작상태이름
%%
/* 규칙절 */
정규표현식        BEGIN 시작상태이름;
<시작상태이름>정규표현식      수행코드;
```

위의 형식처럼 시작 상태를 선언 할 수 있는데 BEGIN으로 시작된 상태는 그 끝에 반드시 BEGIN INITIAL;을 선언하여 원래 기본 상태로 돌아올 수 있도록 해야한다.
이후 토큰이 인식되면 이제 연산파트가 수행된다. 연산파트에는 정의절에 선언한 카운트 변수, 상수외에 Lex에서 기본적으로 제공하는 특별한 전역변수를 사용할 수 있는데 yytext, yyleng 이 있다. 각각 변수는 정규표현식에 의해 실제 인식된 문자열을 갖는 문자배열형 변수, 인식된 문자열의 길이를 갖는 변수이다. 연산파트에서는 이 변수를 활용하여 실제 다양한 연산식을 만들어 낼 수 있는데, 후에 기술 할 문제에서도 유용하게 쓰인다.

3. 서브 루틴절은 Lex에서 사용될 부가적인 코드를 정의하는 파트로 Lex에서 어떠한 처리도 없이 그대로 Lex출력 파일 lex.yy.c 파일에 복사된다. 서브루틴절은 C코드로 이루어지는데 이때 Lex에서 제공하는 Lex함수가 포함된다. 대표적으로 스캐너 역활을 하는 yylex() 가 포함되어야지 말그대로 스캐닝이 이루어지는데 규칙절에 선언한 정규표현과 일치하는 토큰을 찾을 때까지

문자들을 계속 스캐닝하며 토큰번호값들을 반환하게 된다. 이후 파일에서 EOF을 만나 더 이상 읽을 입력이 없으면 yywrap()을 호출 하여 return value를 1로 설정 성공적으로 종료할 수 있게 한다.

2 과제 3번

과제 2-3은 C 코드를 Lex로 분석하여 각 특성에 따라 토큰을 카운팅하여 출력하는 프로그램이다.
먼저 패턴규칙들의 조건을 분석해보면 주석문안의 값은 count 되지 않는다는 조건을 볼 수 있다. 이를 만족하기위해서는 다른 패턴 규칙에서 조건절 안에 있는 문자들을 인식하면 안된다. 이에 따라 위에서 기술했던 조건을 고려해보면 먼저 작성된 규칙이 먼저 인식되므로 주석문의 개수를 세는 규칙이 제일 처음 작성되어야한다. 이를 바탕으로 명시된 규칙대로 순서를 변경하여 test.c 파일과 함께 i xiii 까지의 규칙절을 풀의해볼 수 있다.

```
#include <stdio.h>
#define MAX 987654321
int pem = 01;
int minusnum = -8;
int epm = 2; // epm = 2
/*
pp ppp
*/
int main(){
printf("%d",minusnum++);
int pp[3];
int* ppp = pp;
}
```

2.1 vi

위의 규칙대로 주석문은 명시적인 특성이 있으므로 먼저 작성되어야 한다. 또한 주석문에는 두가지 종류가 있는데 //, /* */ 는 각각 다른 규칙으로 표현했다.
// 는 정규표현식 //.*\n 으로 주석 // 이후에 나오는 모든 문자와 **개행**을 포함하여 인식시켰다.
이후 /* */ 는 시작상태를 정의하여 /*을 만나면 새로운 시작상태를 갖도록 했다.

```
%x DELETE
%%
```

```
...

"/*"     {slash++;}BEGIN DELETE;
<DELETE>.    ;
<DELETE>\n   ;
<DELETE>"./"    BEGIN INITIAL;
```

시작상태 이름을 DELETE로 정의하고 BEGIN DELETE를 통해 상태를 시작했다. 이후 인식되는 문자 및 개행들은 수행코드가 따로 없이 종료된다. 이후 주석이 끝나면 BEGIN INITIAL을 통해 기본상태를 회복한다.

2.2 i

Preprocessor 전처리문의 개수는 #include, #define만 해당하므로 2개이다. 이때 인식되는 문자 토큰은 전처리문이 선언된 행의 전부이다. 따라서 이에 따른 규칙절은 다음과 같이 나타낼 수 있다.

```
#[(include)|(define)].*\n        {preprocessor++;}
```

위의 정규표현식을 해석해보면 #은 공통, [] 안에 include, define중 하나만 골라 인식가능, .* 이후 나오는 모든 문자의 반복0이상, \n개행까지 이에 해당하는 모든 문자열을 인식하는것을 알 수 있다. 이때 여기서 핵심은 개행까지 처리한다는 점이다.

2.3 ii

Octal number 8진수의 개수를 구하려면 8진수의 수의 표현을 먼저 분석해보면 8진수는 0이나온후 0 8중의 숫자가 붙어야 한다. 정규표현식으로 나타내어보면

```
[0-1][0-7] //0~1중 하나, 0~7중 하나
```

[] 를 활용하여 정규표현식을 효과적으로 나타낼 수 있다.

2.4 iii

Negative decimal number 음수인 10진수의 개수 또한 표현을 먼저 분석해보면 -가 먼저 나와야하고 이후 10진수 숫자가 나오면 된다. 이에 따라 정규표현식을 나타내어보면

```
-([0-9])+ //0~9중 고르는걸 1회 이상 반복
```

+기호를 사용하여 숫자가 반복되어 나오는 경우도 나타낼 수 있다.

2.5 iv

Positive decimal number 양수인 10진수를 구하는것이다. iii와 비교해 -연산자만 없으면 된다.

```
([0-9])+
```

2.6 v

명시된 연산자의 개수은 ""Quotation mark를 활용하여 구한다. Quotation mark안에 들어간 모든 문자는 정규표현식의 메타의미를 잃고 문자 그대로 해석되므로 이를 활용하여 정규표현식을 쓰면

```
"+"|"-"|"*"|"/"|"%"      // A 산술연산자
"<"|"<="|"=="|">"|"!="   // B 논리연산자
"||"|"&&"|"!"            // C 관계연산자
"++"|"--"                // D 증감연산자
","                      // E 콤마연산자
"."|"->"                 // F 참조연산자
"&"|"*"                  // G 포인터연산자
```

각 규칙들은 "" 안에 있는 문자들이 문자그대로 인식되고 — 기호에 의해 이들 중 하나일 때의 규칙이 성립하게 된다. 또한 작성된 순서에 따라 처리되므로 연산자 사이의 순서도 구분 할 수 있다.
이를 통해 인식된 문자들은 연산파트의 각 카운트변수들의 값을 올리게 된다.

2.7 vii

= 대입연산자의 개수는 앞의 논리연산자 "=="와 구분을 두어야한다. 둘은 같은 =를 쓰고 있으므로 같은 규칙에 해당 된다. 이를 해결 하기 위해서는 위에 언급한 정규표현식의 선행인식 규칙에 따라 1. "=="논리연산자가 먼저 나와야하고 2. 같은 규칙에 중복 적용될 경우 표현식의 길이가 더 긴것에 먼저 인식되어야 한다. 우리는 이미 논리연산자"=="를 먼저 작성 하였으므로 1번 조건을 만족하여 논리연산자가 모두 인식된 상태이므로 특별한 조건을 더하지 않고 정규표현식을 작성하면된다.

```
"="
```

vi와 같이 ””Quotation mark를 활용하여 기호 문자 그대로를 인식 할 수 있다.

2.8 viii ix

, Bracket의 수 또한 위의 방법과 똑같이 ””Quotation mark를 활용하여 정규표현식을 작성하면

```
"{"     // { 문자 그대로 인식
"}"     // } 문자 그대로 인식
```

위의 정규표현식으로 수행코드 개수 count하면 된다.

2.9 x

Wordcase1 : p가 두번만 들어가는 단어의 개수, 먼저 규칙 조건을 분석해보면 p가 단 두번만 들어가는 단어이어야한다. apple과 같이 p가 중첩되는 단어일 수도 있고, perception과 같이 p가 떨어져 있지만 두 번 들어간 단어이면 인식이되어야 한다. 반면 ppp와 같이 p가 하나라도 많거나 p가 하나만 들어가는 단어는 인식되면 안된다. 이러한 여러 조건들을 보면 정규표현식 만으로는 wordcase1의 경우의 수를 전부 확인 할 수 없다. 설령 [a-zA-z]*p*..[a-zA-Z]*p[a-zA-Z]*과 같은 긴 정규표현식을 구성해도 ppp와 같은 단어 에서는 앞에 pp가 인식되어 같은 단어임에도 뒤의 p하나만 남는 경우가 발생할 수 있다. 또한 **기본적으로 한번 인식된 단어는 다른 조건규칙문 에서 인식할 수 없다는 제약도 존재한다.** 따라서 이런 경우에는 연산파트를 활용 C코드 조건문을 이용하여 case들을 걸러내어야한다.

먼저 단어(word)를 전부 인식시킨다. 이 패턴은 이후 나올 2.10, 2.11까지 공유하는 패턴이다. 모든 단어는 문자열 형태이므로 정규표현식으로 나타내면

```
[a-zA-Z]+       //a~z A~Z중 하나의 문자가 한번이상(+) 반복되는것 인식
```

이어서 인식된 모든 토큰을 연산 파트에서 처리를 할 때, 위에 기술했던 yyleng, yytext 변수를 사용하여 토큰의 value를 직접 이용할 수 있다. yyleng은 문자열의 길이를 저장(정수형), yytext는 문자열을 저장한 배열형이므로 배열과 같이 사용 할 수 있다.

따라서 연산파트를 구성하면 다음과 같이 구성할 수 있다.

```
{for(i = 0; i < yyleng; i++)        //yyleng은 인식된 단어의 길이
    {if(yytext[i] == 'p')           //인식된 문자열 배열의 각 요소를 'p'와 비교
```

```
        {check++;}                  //인식된 문자가 'p'일시 check변수 ++
    };
if(check == 2){wordcase1++;}        //for문 종료 후 check이 2이면 wordcase1 변수++
}
```

이를 통해 배열 요소 중 문자 'p'와 일치하는 요소가 2개인 경우 인식되고 wordcase1에 카운트된다.

2.10 xi

Wordcase2 : 'e'로 시작하고 마지막 글자가 'm'인 단어의 개수, Wordcase1에 이어서 단어의 구성 규칙이 정해져있으므로 같은 방법으로 모든 단어를 인식 한 후에 첫글자가 'e'이고 마지막글자가 'm'인 조건에 맞춰 case를 걸러내어야 한다. 이때 2.9 절 에서도 이미 모든 단어를 인식 했으므로 같은 규칙정의 에서 연산파트만 추가하여 wordcase1에서 인식된 단어를 제외한 후 연산코드를 수행할 수 있다.

이를 효과적으로 수행하기 위해서 2.9절의 연산파트에 $elseif$문을 추가하여 같은 규칙 정의에서 Wordcase2 까지 count할 수 있다.

```
...2.9절의 연산파트에 이어...

 // yytext 문자열 배열의 첫글자index(0) && 마지막글자 index (yyleng-1)
else if(yytext[0] == 'e' && yytext[yyleng -1] == 'm'){

     // 조건에 맞으면 wordcase2 ++
    wordcase2++;
    }
```

이를 통해 배열의 첫글자와 마지막 글자가 조건에 일치하는 단어만 인식되고 wordcase2에 카운트된다.

2.11 xii

Word : 그 외의 남은 단어의 개수, 맨 처음 2.9절에서 모든 단어가 인식되고 이제 남은건 Wordcase1, Wordcase2에 걸러지지 않은 나머지 단어뿐이다.
남은 단어들은 같은 패턴 연산 파트에 $else$문 만 추가하여 처리 할 수 있다.

```
...2.9 절의 연산파트 ...
```

```
...2.10절의 연산파트 ...

else {
    word++;     // case조건에 포함되지 않고 남은 단어가 인식될 시 word++
    check = 0;  // if(yytext[i] == 'p') 단어에서 p가 하나라도 있었을 시 초기화
}
```

마지막 line의 $check = 0$;은 2.9절의 연산파트와 밀접한 관련이 있다. 단편적인 예를 들어보면 만약 peace라는 단어를 패턴토큰이 인식을 했다 가정했을 때, peace는 문자 'p'가 하나이므로 check 변수의 value는 1이 된다. 이는 어떠한 조건문에도 해당되지않아 else문까지 오게된다. 이때 다음 단어토큰의 연산 수행에 앞서 check 변수의 value를 0으로 초기화 시키기 위함이다.

2.12 xiii

위에서 카운트 되지 않은 문자의 개수, 2.1 2.11에 걸쳐서 인식된 모든 문자들을 제외한 test.c input 값에서 공백, 기호, 개행 등을 카운트하면 된다. 남은 모든 것들을 인식하기 위한 정규표현식은

```
.|\n        {mark++;} // . 모든 문자와 개행문자를 인식
```

위와 같은 정규표현식으로 이전까지 인식되지 않은 모든 문자, 공백, 개행이 인식되고 이후 연산파트가 수행되며 변수 mark가 카운트된다.

2.13 서브루틴절

main()함수 안에 yylex()함수가 실행되므로서 각 입력을 하나하나 읽어나가게 되고 정규표현이 일치하는 토큰이 있으면 결합된 연산파트 수행코드를 실행하게 된다. 모든 input이 인식되고 EOF에 의해 yylex()가 yywrap()을 호출하여 종료하게 되면 printf()함수에 의해 각 카운팅 변수들이 출력되게된다.

```
%%
int main(){
    yylex();
    printf("preprocessor : %d\n", preprocessor);
```

```
    ...중략...
    printf("mark : %d\n", mark);
    return 0;       //main함수의 종료
}

int yywrap(){
    return 1;   // EOF 만났을 때 정상적인 종료
    }
```