프로그래밍 언어 HW3: Yacc

B743014 양혜진

April 30, 2021

1 Yacc의 동작 방식

Lex란, Lexical Analyzer Generator 의 준말로, 어휘 분석기(lexical analyzer) 를 생성하기 위해 사용하는 도구이다. 기술적으로 Lex는 정규표현식의 모음으로 표현된 Lex 소스코드(".l" 확장자로 끝나는 파일)를 C 코드의 구현(lex.yy.c 파일)으로 변환해준다. 이렇게 생성된 C 프로그램을 컴파일하여 실행 가능한 어휘 분석기(lexical analyzer)를 생성할 수 있다. 이렇게 생성된 어휘 분석기의 입력으로 파일을 넣어주면, lex 코드에 정의된 규칙에 따라 파일을 분석하고, 결과를 출력한다. lex 에 의해 생성되는 함수들 중 과제에 사용한 함수는 다음과 같다

- **yylex() 함수** : lex 소스코드를 C 코드 구현으로 변환하기 위해 사용하는 lex 명령어는는 규칙절에 작성된 규칙에 대한 C 코드를 생성하고, 이것을 yylex()라는 단일 함수로 만들어 lex.yy.c 파일에 저장한다. 생성된 lex.yy.c 파일은 cc 명령어를 통해 스캐너를 호출할 수 있는 실행 가능한 프로그램으로 컴파일할 수 있다. 만약 yylex ()가 입력 스트림의 문자열과 일치하면 규칙 섹션에서 정의된 작업을 실행하기 전에 일치하는 텍스트 부분을 yytext 라는 외부 문자 배열로 복사한다. 그 다음에 규칙절에 작성된 규칙에 대한 C코드가 실행된다.
- **yywrap() 함수** : 입력이 소진되었거나, EOF 에 다다랐을 때 lex 에 의해 호출되는 함수이다. 즉, 입력이 종료되었을 때 호출되는 함수이다.

2 hw3.l

- 파일명 : hw3.l
- 내용 : 텍스트에서 'love'라는 단어가 몇 번 나오는지 카운트하는 lex코드 작성

2.1 hw2_1.l 소스 코드

```
%{
#include <stdio.h>
int lovecount = 0;
%}

%%
(L|l)ove {lovecount++;}
.|\n  ;
%%

int main() {
   yylex();
   printf("number of love=%d\n", lovecount);
```

```
   return 0;
}

int yywrap() {
   return 1;
}
```

2.2 hw3.l 코드 분석

- 정의절(Definitions Section)

```
%{
#include <stdio.h>
int lovecount = 0;
%}
```

정의절은 %{ 와 %{ 사이에 작성하며, 최종적으로 컴파일될 Lex 프로그램에 포함하고자하는 C code 를 포함한다. 일반적으로 #include 나 #define 과 같은 전처리문이나 전역변수 선언이 들어간다. hw2_1.l 에서는 love 의 개수 출력을 위해 printf() 함수가 필요하므로 stdio.h 헤더가 포함되어야 한다. 그래서 정의절에 #include <stdio.h> 전처리문을 작성하였다. 또한, 단어 love 를 찾을 때마다 count 되는 개수를 저장하기 위해서 정수형 전역변수 lovecount 를 선언하고, 값을 0으로 초기화해주었다.

- 규칙절(Rules Section)

```
(L|l)ove {lovecount++;}
.|\n  ;
```

규칙절은 찾고자 하는 문자열 패턴의 규칙을 정규표현식(Regular Expression)을 사용해서 작성한다. 정의절과 규칙절은 %% 로 구분된다. 과제의 love.txt 파일에 있는 love 는 앞 글자의 대소문자에 상관없이 패턴이 매칭되어야 하기 때문에 (L|l)ove 로 규칙을 작성해주었다. (L|l)에서 괄호()는 정규표현식에서 그룹핑을 의미한다. 그룹핑은 정규표현식 패턴 매칭에서의 우선순위를 의미하며, L 또는 l 을 의미한다. 따라서 (L|l)ove 에 매칭되는 문자열은 Love 또는 love 이다. 또한, 해당 규칙과 매칭되는 문자열이 있는 경우 수행할 C 코드를 작성할 수 있는데, 정의절에서 선언한 전역변수 lovecount 의 값을 1 증가시키도록 하였다.

- 사용자 서브루틴절(User Subroutines Section)

```
int main() {
   yylex();
   printf("number of love=%d\n", lovecount);
   return 0;
}

int yywrap() {
   return 1;
}
```

사용자 서브 루틴절은 코드절(Code Section)이라고도 한다. 이곳에서는 main() 함수 안에 yylex()와 같은 lex 의 기본 서브루틴을 호출하거나, 재정의할 수 있다. 입력 스트림에 대해 규칙을 수행하기 위해 main() 함수 안에서 yylex() 함수를 호출해주도록 했다. 또한, EOF가 되어 yylex() 함수가 종료되면 지금까지 규칙절에 작성한 것에 따라 카운트한 love

의 개수를 print() 함수로 출력하도록 했다. 또한, yywrap() 함수를 재정의하여 입력이 소진되거나 EOF 에 다다르면 1을 반환하도록 하였다.

3 과제2: DANGEROUS PATTERN

- 파일명 : hw2_2.l
- 내용 : Python 과제 6번을 lex 코드로 작성 (문자열의 개수 입력 받지 않음)
  * Python 과제 6번 : (100~1~|01)~을 만나면 is danger 출력

3.1 hw2_2.l 소스 코드

```
%{
#include <stdio.h>
%}

%%
^(100+1+|01)+$ {printf("%s is danger\n", yytext);}
.|\n  ;
%%

int main() {
   yylex();
   return 0;
}

int yywrap() {
   return 1;
}
```

3.2 hw2_2.l 코드 분석

- 정의절(Definitions Section)

```
%{
#include <stdio.h>
%}
```

hw2_2.l 에서는 pattern 의 위험여부 출력을 위해 printf() 함수가 필요하므로 stdio.h 헤더가 포함되어야 한다. 그래서 정의절에 #include <stdio.h> 전처리문을 작성하였다.

- 규칙절(Rules Section)

```
^(100+1+|01)+$ {printf("%s is danger\n", yytext);}
.|\n  ;
```

과제에서 찾을 암호화되지 않은 데이터의 패턴은 정규표현식 ^(100+1+|01)+$ 로 작성했다. ^(100+1+|01)+ 에서 괄호 ()는 정규표현식에서 그룹핑을 의미한다. 그룹핑은 정규표현식 패턴 매칭에서의 우선순위를 의미하며, 괄호 안의 의미는 100+1+ 또는 01 이라는 뜻이다. 또한, 어떤 문자 뒤의 + 기호는 해당 문자가 한 번 이상 반복되어야 한다는 뜻이므로 10010 또는 100011 과 같이 + 앞의 문자 한 개가 한 번 이상 반복될 수

있다. 괄호 () 앞의 ^ 기호는 해당 패턴으로 시작된다는 의미이므로, 100+1+ 또는 01로 시작하는 패턴이어야 하며, 정규표현식 맨 뒤의 $ 기호는 해당 패턴의 문자 다음이 개행이라는 뜻이므로, 한 줄의 모든 문자열이 (100+1+|01)+ 패턴과 일치해야지만 해당 데이터 패턴이 위험하다는 내용의 문자열 (%s is danger) 출력할 수 있다.

- 사용자 서브루틴절(User Subroutines Section)

```
int main() {
   yylex();
   return 0;
}

int yywrap() {
   return 1;
}
```

사용자 서브 루틴절에서는 main() 입력 스트림에 대해 규칙을 수행하기 위해 main() 함수 안에서 yylex() 함수를 호출해주도록 했다. 또한, 패턴과 일치하는 형태의 암호화되지 않은 문자열일 경우, 규칙절에 작성된 규칙에 따라 yylex() 안에서 해당 패턴이 위험하다는 문구가 출력되므로, main() 함수 내에서 yylex() 함수 외에 다른 함수를 호출하거나 작성할 필요가 없어 다른 기능을 작성하지 않았다. 또한, yywrap() 함수를 재정의하여 입력이 소진되거나 EOF 에 다다르면 1을 반환하도록 하였다.

4 과제3: C Code Analyzer

- 파일명 : hw2_3.l
- 내용 : C 소스코드를 읽고 분석하여 코드에 나오는 특성에 따라 카운트한다.

4.1 hw2_3.l 소스 코드

```
%{
#include <stdio.h>
int preprocessor = 0;
int octal_number = 0;
int negative_decimal_number = 0;
int positive_decimal_number = 0;
int operator = 0;
int comment = 0;
int equal_sign = 0;
int lhs_bracket_sign = 0;
int rhs_bracket_sign = 0;
int wordcase1 = 0;
int wordcase2 = 0;
int word = 0;
int mark = 0;
%}

DIGIT [0-9]
WORD [0-9A-Za-z_]
WORD_START [A-Za-z_]
WORD_NOT_P [0-9A-OQ-Za-oq-z_]
WORD_NOT_P_START [A-OQ-Za-oq-z_]
```

```
%%
#(.*)\n {preprocessor++;}
0[1-7][0-7]* {octal_number++;}
-{DIGIT}+ {negative_decimal_number++;}
{DIGIT}+ {positive_decimal_number++;}
&{2} {operator++;}
\|{2} {operator++;}
(=|!|>|\<)= {operator++;}
\+{2} {operator++;}
\-{2} {operator++;}
! {operator++;}
> {operator++;}
\< {operator++;}
, {operator++;}
& {operator++;}
\* {operator++;}
(\+|\-|\*|\/|%) {operator++;}
"//"(.*)\n {comment++;}
"/*"([^\*]|(\*+[^\/]))*"*/" {comment++;}
= {equal_sign++;}
\{ {lhs_bracket_sign++;}
\} {rhs_bracket_sign++;}
{WORD_NOT_P_START}*{WORD_NOT_P}*p{WORD_NOT_P}*p{WORD_NOT_P}* {wordcase1++;}
e{WORD}*m {wordcase2++;}
{WORD_START}{WORD}* {word++;}
\n|. {mark++;}
%%

int main() {
   yylex();
   printf("preprocessor = %d\n", preprocessor);
   printf("octal number = %d\n", octal_number);
   printf("negative decimal number = %d\n", negative_decimal_number);
   printf("positive decimal number = %d\n", positive_decimal_number);
   printf("operator = %d\n", operator);
   printf("comment = %d\n", comment);
   printf("'=' = %d\n", equal_sign);
   printf("'{' = %d\n", lhs_bracket_sign);
   printf("'}' = %d\n", rhs_bracket_sign);
   printf("wordcase1 = %d\n", wordcase1);
   printf("wordcase2 = %d\n", wordcase2);
   printf("word = %d\n", word);
   printf("mark = %d\n", mark);
   return 0;
}

int yywrap() {
   return 1;
}
```

4.2 hw2_3.l 코드 분석

- 정의절(Definitions Section)

```
%{
#include <stdio.h>
int preprocessor = 0;
int octal_number = 0;
int negative_decimal_number = 0;
int positive_decimal_number = 0;
int operator = 0;
int comment = 0;
int equal_sign = 0;
int lhs_bracket_sign = 0;
int rhs_bracket_sign = 0;
int wordcase1 = 0;
int wordcase2 = 0;
int word = 0;
int mark = 0;
%}
```

hw2_3.l 카운트한 C 소스코드 특성들의 개수 출력을 위해 printf() 함수가 필요하므로 stdio.h 헤더가 포함되어야 한다. 그래서 정의절에 #include <stdio.h> 전처리문을 작성하였다. 또한 카운트할 각각의 요소들의 개수를 저장할 정수형 전역 변수들을 추가해주었다.

  - preprocessor : 전처리문의 개수
  - octal number : 8진법 숫자의 개수
  - negative decimal number : 10진법 숫자 중 음수의 개수
  - positive decimal number : 10진법 숫자 중 양수의 개수
  - operator : 산술, 논리, 관계, 증감, 콤마, 참조, 포인터 연산자의 개수
  - comment : 주석문(signle line, multi line)의 개수
  - equal sign : '=' 기호(대입 연산자)의 개수
  - lhs bracket sign : '' 기호의 개수
  - rhs bracket sign : '' 기호의 개수
  - wordcase1 : p 가 두 개만 들어간 단어의 개수
  - wordcase2 : e 로 시작하고 마지막 글자가 m 인 단어의 개수
  - word : 그 외 단어의 개수
  - mark : 위에서 count 되지 않은 문자의 개수

- 추가적인 매크로 정의절(optional macro definitions for regular expressions)

```
DIGIT [0-9]
WORD  [0-9A-Za-z_]
WORD_START [A-Za-z_]
WORD_NOT_P [0-9A-OQ-Za-oq-z_]
WORD_NOT_P_START [A-OQ-Za-oq-z_]
```

hw2_3.l 의 규칙절에서 정규표현식의 패턴을 단순화하기 위해 사용할 변수를 선언했다. 변수를 너무 많이 사용하면 오히려 코드의 길이가 길어지고 가독성이 낮아질 수 있으므로, 길고 복잡한 패턴이나 두 번 이상 반복되는 패턴에 대해서만 변수를 선언하였다.

- DIGIT : 0 과 9 사이의 숫자
- WORD : 0 ∼ 9 또는 영어 대소문자 또는 언더바(under bar, _)
- WORD_STRART : 영어 대소문자 또는 언더바(under bar, _)
- WORD_NOT_P : 0 ∼ 9 또는 p 와 P 를 제외한 영어 대소문자 또는 언더바(under bar, _)
- WORD_NOT_P : p 와 P 를 제외한 영어 대소문자 또는 언더바(under bar, _)

• 규칙절(Rules Section)

```
#(.*)\n {preprocessor++;}
0[1-7][0-7]* {octal_number++;}
-{DIGIT}+ {negative_decimal_number++;}
{DIGIT}+ {positive_decimal_number++;}
&{2} {operator++;}
\|{2} {operator++;}
(=|!|>|\<)= {operator++;}
\+{2} {operator++;}
\-{2} {operator++;}
! {operator++;}
> {operator++;}
\< {operator++;}
, {operator++;}
& {operator++;}
\* {operator++;}
(\+|\-|\*|\/|%) {operator++;}
"//"(.*)\n {comment++;}
"/*"([^\*]|(\*+[^\/]))*"*/" {comment++;}
= {equal_sign++;}
\{ {lhs_bracket_sign++;}
\} {rhs_bracket_sign++;}
{WORD_NOT_P_START}*{WORD_NOT_P}*p{WORD_NOT_P}*p{WORD_NOT_P}*
    {wordcase1++;}
e{WORD}*m {wordcase2++;}
{WORD_START}{WORD}* {word++;}
\n|. {mark++;}
```

lex 의 규칙절은 위에서부터 순서대로 수행되므로, 동일한 문자가 1개 있는 패턴과 2개 있는 패턴이 있는 경우 문자 개수가 많은 것부터 차례대로 검사해 주어야 한다. 예를 들어 == 패턴과 = 패턴을 다른 패턴으로 구별해야 하는 경우, = 패턴을 먼저 검사하고 == 패턴을 그 다음에 검사하면, == 패턴을 = 패턴이 2개 있는 것으로 인식하기 때문에 원하는 결과를 얻을 수 없다.

규칙절에 기술된 패턴과 매칭되는 문자열을 찾은 경우, 옆의 {} 안에 작성된 코드를 실행하게 된다. 그래서 일치하는 C 언어 요소의 정규표현식에 따라 각 요소의 카운트를 증가연산자로 1 증가해주도록 코드를 작성하였다.

- **전처리기문 정규표현식 및 규칙**

```
#(.*)\n {preprocessor++;}
```

특수문자 #으로 시작한 뒤, 개행\ n을 만나기 전까지의 모든 문자열의 반복(.)을 의미한다.

- **8진법 숫자 정규표현식및 규칙**

```
0[1-7][0-7]* {octal_number++;}
```

숫자 0으로 시작한뒤, 바로 뒤에 0이 아닌 1 과 7 사이의 숫자가 한 개 오고, 그 다음부터는 0과 7 사이의 숫자가 0번 이상 반복하여 이루어진 문자열을 의미한다.

- **10진법 숫자 음수 정규표현식 및 규칙**

```
-{DIGIT}+ {negative_decimal_number++;}
```

기호 로 시작한 뒤, 0 과 9 사이의 숫자가 한 번 이상 반복하여 이루어진 문자열을 의미한다. 10진수 음수 숫자를 10진수 양수 숫자보다 먼저 검사해야한다. 만약 10진수 양수 숫자를 10진수 음수 숫자보다 먼저 검사하면, 10 과 같은 수에서 를 제외하고 10 을 10진수 양수의 패턴에 맞는 것으로 인식하기 때문이다.

- **10진법 숫자 양수 정규표현식 및 규칙**

```
{DIGIT}+ {positive_decimal_number++;}
```

0 과 9 사이의 숫자가 한 번 이상 반복하여 이루어진 문자열을 의미한다.

- **연산자 정규표현식 및 규칙**

```
&{2} {operator++;}
\|{2} {operator++;}
(=|!|>|\<)= {operator++;}
\+{2} {operator++;}
\-{2} {operator++;}
! {operator++;}
> {operator++;}
\< {operator++;}
, {operator++;}
& {operator++;}
\* {operator++;}
(\+|\-|\*|\/|%) {operator++;}
```

연산자의 경우에도 동일한 문자가 1개 있는 패턴과 2개 있는 패턴을 구분해야하므로, 문자가 2개 있는 패턴부터 검사해주도록 했다. 몇 개의 문자앞에 있는 백슬래시 기호들은 해당 기호들이 본래 정규표현식에서 사용되는 메타(Meta) 문자이기 때문에, 메타 문자가 의미하는 특수한 의미가 아닌 해당 문자 자체를 사용하기 위해 붙여주었다.

- **주석 정규표현식 및 규칙**

```
"//"(.*)\n {comment++;}
"/*"([^\*]|(\*+[^\/]))*"*/" {comment++;}
```

큰 따옴표는 백슬래시를 사용하지 않고 메타 문자를 문자 자체로 인식하기 위해 사용한다. 첫 번째 정규표현식은 single line comment 의 정규표현식으로, 슬래시 문자 두개가 먼저 나오고, 개행이 등장하기 직전까지의 모든 문자열의 반복을 의미한다. 두 번째 정규표현식은 multi line comment 의 정규표현식으로, 슬래시와 별표(asterisk) 문자가 먼저 나오고, 그 뒤로 별 표시가 아닌 모든 문자 또는 별표시가 한 개 이상 반복되고 그 뒤에 개행이 아닌 모든 문자가 0 번 이상 반복된 뒤 별표시와 슬래시 문자로 끝나는 문자열 패턴이다. 이것은 별표시가 multi line comment 사이에 들어오기 위해서는 별표시 뒤에 슬래시가 있으면 안된다는 의미이다. 또한, multi line comment 는 single line comment 와 달리 패턴 끝에 개행문자가 포함되지 않는다.

– **대입연산자 정규표현식 및 규칙**

```
= {equal_sign++;}
```

– **'{'기호 정규표현식 및 규칙**

```
\{ {lhs_bracket_sign++;}
```

– **'}'기호 정규표현식 및 규칙**

```
\} {rhs_bracket_sign++;}
```

– **wordcase1 정규표현식 및 규칙**

```
{WORD_NOT_P_START}*{WORD_NOT_P}*p{WORD_NOT_P}*p{WORD_NOT_P}*
    {wordcase1++;}
```

p 가 두 개만 들어가는 단어의 규칙이다. 여기서 단어(word)는 C 언어의 변수 명명 규칙에 따라 첫 글자는 숫자를 제외한 영어 대소문자 또는 언더 바(under bar)이고, 그 뒤의 글자는 숫자와 영어 대소문자 또는 언더 바(under bar)들의 반복으로 이루어진 문자열을 의미한다. 따라서 처음 글자는 WORD_NOT_P_START 변수에 선언한 p 를 제외한 영어 대소문자 또는 언더 바(under bar)가 0번 이상 반복되고, 그 다음으로 나오는 문자 p 사이에 WORD_NOT_P 변수에 선언한 숫자 또는 p 를 제외한 영어 대소문자 또는 언더 바(under bar)가 0번 이상 반복되는 문자열의 패턴으로 작성하였다.

– **wordcase2 정규표현식 및 규칙**

```
e{WORD}*m  {wordcase2++;}
```

e 로 시작하고, m 으로 끝나는 단어의 규칙이다. 가운데에 있는 WORD 변수는 숫자 또는 영어 대소문자 또는 언더 바 문자를 의미한다.

– **word 정규표현식 및 규칙**

```
{WORD_START}{WORD}* {word++;}
```

word 는 C 언어의 변수 명명 규칙과 동일하다. 맨 처음 글자는 숫자가 아닌 영어 대소문자 또는 언더 바로 이루어져 있으며, 그 다음으로 숫자 또는 영어 대소문자 또는 언더 바 문자가 0 번이상 반복하여 이루어진 문자열 패턴을 의미한다.

– **mark 정규표현식 및 규칙**

```
\n|. {mark++;}
```

위의 정규표현식과 매칭되지 않은 나머지 문자들을 세어주는 규칙이다. $\backslash n$ 는 개행문자를 의미하고, .은 개행문자를 제외한 모든 문자를 의미한다. 즉 위의 정규표현식과 매칭되지 않았으면 모두 이 규칙에 해당하게 된다.

- 사용자 서브루틴절(User Subroutines Section)

```
int main() {
   yylex();
   printf("preprocessor = %d\n", preprocessor);
   printf("octal number = %d\n", octal_number);
   printf("negative decimal number = %d\n", negative_decimal_number);
   printf("positive decimal number = %d\n", positive_decimal_number);
   printf("operator = %d\n", operator);
   printf("comment = %d\n", comment);
   printf("'=' = %d\n", equal_sign);
   printf("'{' = %d\n", lhs_bracket_sign);
   printf("'}' = %d\n", rhs_bracket_sign);
   printf("wordcase1 = %d\n", wordcase1);
   printf("wordcase2 = %d\n", wordcase2);
   printf("word = %d\n", word);
   printf("mark = %d\n", mark);
   return 0;
}

int yywrap() {
   return 1;
}
```

사용자 서브 루틴절에서는 main() 입력 스트림에 대해 규칙을 수행하기 위해 main() 함수 안에서 yylex() 함수를 호출해주도록 했다. 또한, yylex() 함수가 종료된 뒤, yylex() 함수에서 카운트한 C 소스코드 요소들의 개수를 출력해주어야 하기 때문에 전역 변수로 선언된 요소들의 개수 변수를 출력할 수 있는 printf()문을 작성하였다. 그리고 yywrap() 함수를 재정의하여 입력이 소진되거나 EOF 에 다다르면 1을 반환하도록 하였다.