기초컴파일러구성

Project 보고서

20161663

허재성

예제 13-1

이론적 내용

C에서의 식별자는 처음에 알파벳 대소문자 또는 _로 시작하면서 알파벳, _, 숫자를 포함한다. 한편 렉스에서 입력 파일 형식은

```
정의 부분
%%
변환 규칙 부분
%%
사용자 부프로그램 부분
```

으로 되어있으며 정의 부분과 사용자 부프로그램 부분은 생략 가능하지만 변환 규칙 부분은 필수로 존재해야 한다.

정의 부분은 이름과 일련의 표현식으로 구성되며 이름은 식별자이고 표현식은 이름에 해당하는 정규 표현식이다. 이를 토대로 C에서의 식별자에 대한 정의 부분을 작성해보면 다음과 같다. 알파벳과 _를 Letter, 0~9의 숫자를 Digit이라 하면

```
Letter [a-zA-Z_]
Digit [0-9]
```

로 나타낼 수 있다. 변환 규칙 부분은 표현식과 수행코드로 이루어진다. 표현식은 정규 표현이고, 수행 코드는 표현식이 나타낸 정규 표현과 매칭되었을 때 수행될 코드로 C로 작성되어 있다.정의 부분에서 정의한 Letter와 Digit을 이용해 변환 규칙 부분을 작성하면 다음과 같다.

```
{Letter}({Letter} | {Digit})* return tident
```

{Letter}({Letter} | {Digit})*에서 {Letter}는 식별자의 첫 문자가 정의 부분에서 정의한 Letter로 시작함을 나타내며 ({Letter} | {Digit})*은 Letter 또는 Digit으로 이루어진 문자열을 나타낸다. 이 때 *은 해당 내용이 0번 이상 반복됨을 나타낸다. 즉 첫 문자만 Letter로 존재할 수도 있다.

결과

따라서 렉스 입력은 다음과 같다. 정의 부분과 변환 규칙 부분을 %%로 구분하였다.

```
Letter [a-zA-Z_]
Digit [0-9]
%%
{Letter}({Letter} | {Digit})* return tident
```

예제 13-2

이론적 내용

이론적 내용은 13-1과 크게 다르지 않다. 차이는 숫자를 표현하려면 0-9 중 하나가 적어도 한번은 쓰여야 하므로 0번 이상의 반복이 아닌 최소 한번 이상의 반복을 표현해야 한다. 이는 * 대신 +를 사용하면 된다.

결과

따라서 변환 규칙 부분은 다음과 같다.

```
[0-9]+ printf("found integer number₩n");
```

예제 13-3

이론적 내용

야크의 입력 파일 형식은 렉스와 비슷하게 다음과 같이

```
선언 부분
%%
변환 규칙 부분
%%
사용자 프로그램 부분
```

로 구성되어 있다.

선언 부분에서는 문법 규칙과 토큰에 대해 선언하며 변환 규칙 부분과 프로그램 부분에서 사용될 임시 변수에 대해 선언한다.

야크의 변환 규칙 부분에서는 문법의 규칙을 정의하며 각 규칙은 Left Hand Side(LHS)와

Body로 구성된다. LHS는 논터미널의 이름이고 Body는 문법 규칙의 오른쪽 부분에 나오는 기호로 이름 또는 상수 문자가 될 수 있다.

결과

BNF로 표현된 규칙 <expr>::=<expr> + <expr>은 야크로

```
expr : expr'+'term
```

으로 나타낼 수 있다. 이 때 + 같은 operator는 ''로 표현한다.

예제 13-4

이론적 내용

모호한 문법을 모호하지 않은 문법으로 만들기 위해서 연산 순서를 확실히 정의하고 또한 연산 시 결합 법칙(왼쪽 결합법칙과 오른쪽 결합법칙)을 정해야 한다. 야크에서는 이를 위해 우선순위를 제공하는데 가장 낮은 우선순위의 터미널(연산자)부터 정의하여 위에 놓고, 가장 높은 우선순위의 연산자는 아래에 놓는다. 또한 연산자가 왼쪽 결합법칙을 따를 경우 LEFT로, 오른쪽 결합법칙을 따를 경우 RIGHT로 표기한다..

결과

```
%right'='
%leftt'+"-'
%left'*','/'
```

선언 부분을 보면 연산자 우선 순위는 *,/ > +,- > = 이고 =는 오른쪽 결합법칙을 따르며 +,-,*,/ 는 왼쪽결합 법칙을 따르는 것을 알 ㅅ n있다.

따라서 a = c * d – e + f / g 를 연산 순서와 결합법칙을 고려해 괄호로 묶어 표현하면

```
(a = (((c * d) – e) + (f / g)))
```

이 된다.

예제 13-5

이론적 내용

렉스 입력 파일의 정의 부분에 자료구조, 변수, 상수 등을 정의하기 위해서는 가장 위에 %{로 시작하여 %}로 끝나는 부분에 해당 내용을 기술하면 된다. 실행 결과 생성되는 lex.yy.c 소스 파일에 그대로 복사되는 부분이다. 사용자 부프로그램 부분에는 main 함수를 정의하여 렉스 파일로부터 만들어진 lex.yy.c에서 yylex 함수를 호출하여 실행한다. 사용자 부프로그램의 main 부분은 lex.yy.c 파일의 가장 아래에 그대로 복사된다. 이를 이용해 렉스 입력 파일을 작성하면 다음과 같다.

소스 코드 (WordCount.l)

```
%{
/*
 word count
*/
unsigned charCount=0, wordCount=0, lineCount=1;
%}

word [^ \t\n]+
eol \n

%%

{word} {wordCount++; charCount+=yyleng; printf("word : %s\n", yytext);}
{eol}  {charCount++; lineCount++;}
.      charCount++;

%%
main()
{
    yylex();
    printf("number of character : %d\n", charCount);
    printf("number of word : %d\n", wordCount);
    printf("number of line : %d\n", lineCount);
}
```

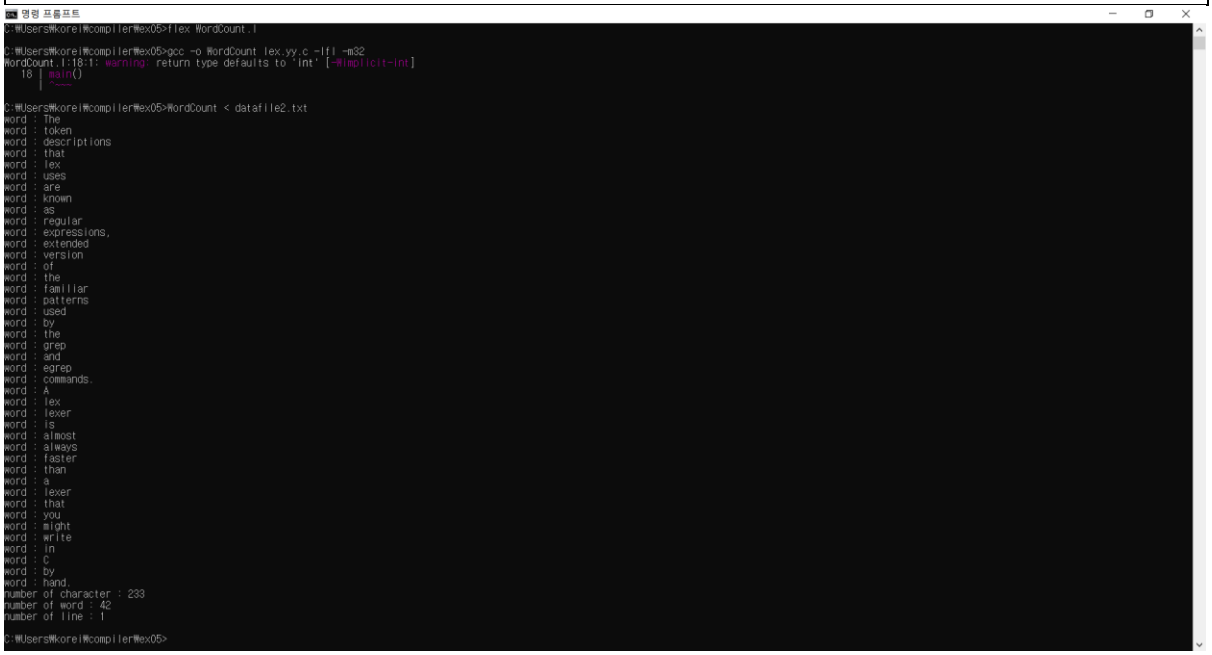
WordCount.l을 실행하면 lex.yy.c 파일을 생성한다. lex.yy.c가 어휘 분석기 프로그램의 소스코드이다. 이를 컴파일하여 실행 파일 WordCount을 생성한 후 실행시키면 어휘 분석이 이루어 지는 것을 알 수 있다.

컴파일 시 주의사항으로 플렉스 라이브러리를 링크시키기 위해 -lfl 플래그를 추가시켜야

하며 x64에서 실행할 경우 32bit 라이브러리에 링크하기 위해 -m32 플래그를 추가해야 한다.

실행 과정 및 결과는 다음과 같다. 어휘 분석에 쓰인 text는 datafile2.txt에 저장되어 있으며 내용은 다음과 같다.

```
The token descriptions that lex uses are known as regular expressions, extended version of the familiar patterns used by the grep and egrep commands. A lex lexer is almost always faster than a lexer that you might write in C by hand.
```



어휘가 띄어쓰기 기준으로 잘 구분된 것을 알 수 있다.

예제 13-6

이론적 내용

토큰 별로 구분하는 어휘 분석기를 만들기 위해 토큰을 정의해야 한다. 먼저 주어진 텍스트(datafile.txt)를 보면

```
ni = ba * po - 60 + ni / (abc + 50);
```

에서 ni, ba, po, abc, 60, 50, *, +, -, / (,), ;가 토큰임을 알 수 있다. Ni, ba, po, abc는 식별자이므로 식별자의 정규 표현식을, 60, 50은 수이므로 숫자를 이용한 정규 표현식을 이용하여 표현하고, +, -, *, /, (,), ;는 그 자체로 토큰이 되므로 각각을 변환 규칙 부분에 표현해주어 해당하는 것을 return해주면 된다. Return을 위해 정의 부분에 각 토큰에 해당하

는 이름을 열거형으로 정의해 둔다. 이 때 주의할 점은 각 토큰이 띄어쓰기에 의해 구분되므로 띄어쓰기 ' '가 나타날 때 마다 특별한 return이 없도록 하면 띄어쓰기를 만날 때 마다 별 다른 행동 없이 다음 토큰을 찾는 것을 알 수 있다. 마지막으로 사용자 부프로그램 부분에 각 토큰을 만났을 때의 출력문을 작성한다.

소스 코드 (test.l)

```
%{ /*정의 부분*/
#include <stdio.h>
#include <stdlib.h>
enum tnumber {TEOF, TIDEN, TNUM, TASSIGN, TADD,
              TSUBT, TMUL, TDIV, TSEMI, TBEGIN,
              TEND, TERROR};
%}

/* 이름 n 치환식 */
letter[A-Za-z]
digit[0-9]

%% /*정의 부분*/
"("                                  return(TBEGIN);
")"                                  return(TEND);
{letter}({letter}|{digit})*           return(TIDEN);
=                                    return(TASSIGN);
"+"                                  return(TADD);
"*"                                  return(TMUL);
"/"                                  return(TDIV);
"-"                                  return(TSUBT);
{digit}+                             return(TNUM);
";"                                  return(TSEMI);
[ \t\n]                               ;
.                                    return(TERROR);
%%/* 사용자 부프로그램 부분 */
main()
{
enum tnumber tn;    /* token number */
printf("Start of FLEX\n");

while((tn=yylex()) != TEOF) switch(tn) {
        case TBEGIN     :printf("Left parenthesis\n");  break;
        case TEND       :printf("Rightt parenthesis\n");    break;
        case TIDEN      :printf("Identifier : %s\n", yytext);   break;
        case TASSIGN    :printf("Assign_op\n"); break;
        case TADD       :printf("Add_op\n");    break;
        case TMUL       :printf("Mul_op\n");    break;
        case TDIV       :printf("Div_op\n");    break;
```

```
        case TSUBT      :printf("Subt_op\n");    break;
        case TNUM       :printf("Number : %s\n", yytext);   break;
        case TSEMI      :printf("Semicolon\n"); break;
        case TERROR     :printf("Error\n"); break;
    }
yywrap();   {printf("End of FLEX\n"); return 1; }
}
```

실행과정은 13-5와 같다.

datafile.txt의 내용을 손으로 직접 분석하면 다음과 같다.

ni
=
ba
*
po
-
60
+
ni
/
(
abc
+
50
)
;

플렉스를 통해 생성한 어휘분석기가 분석한 내용은 다음과 같다. 손으로 한 것과 다르게 (결과는 같지만) 각 토큰이 식별자인지, 연산자인지, 상수인지 등을 추가로 알 수 있다.

Start of FLEX (어휘 분석 시작)
Identifier : ni (식별자 ni)
Assign_op (= 연산자)
Identifier : ba
Mul_op (* 연산자)
Identifier : po
Subt_op (- 연산자)
Number : 60 (상수)
Add_op (+ 연산자)
Identifier : ni
Div_op (/ 연산자)
Left parenthesis (좌괄호 '(')
Identifier : abc
Add_op
Number : 50
Right parenthesis (우괄호 ')')
Semicolon (;)
End of FLEX (어휘 분석 끝)

손으로 분석한 결과와 같은 것을 알 수 있다.

예제 13-7

이론적 내용

해당 문법을 바이슨 파일 tt.y로 만든다. 이 때 터미널 기호를 토큰 ID로 한다. 변환 규칙에 존재하는 S, L, R은 논터미널 기호이며 곱셈 연산자 *는 터미널 기호임을 밝히기 위해 '*'로 나타낼 수 있다.

바이슨에서 생성된 파서 함수는 yyparse로 이름이 고정되어 있다. 바이슨 파일의 사용자 프로그램 부분에는 yyparse의 리턴 값을 확인하여 0일 경우 Parsing이 성공함을, 0이 아닐 경우 문법이 오류가 있어서 Parsing이 실패함을 알린다.

소스 코드

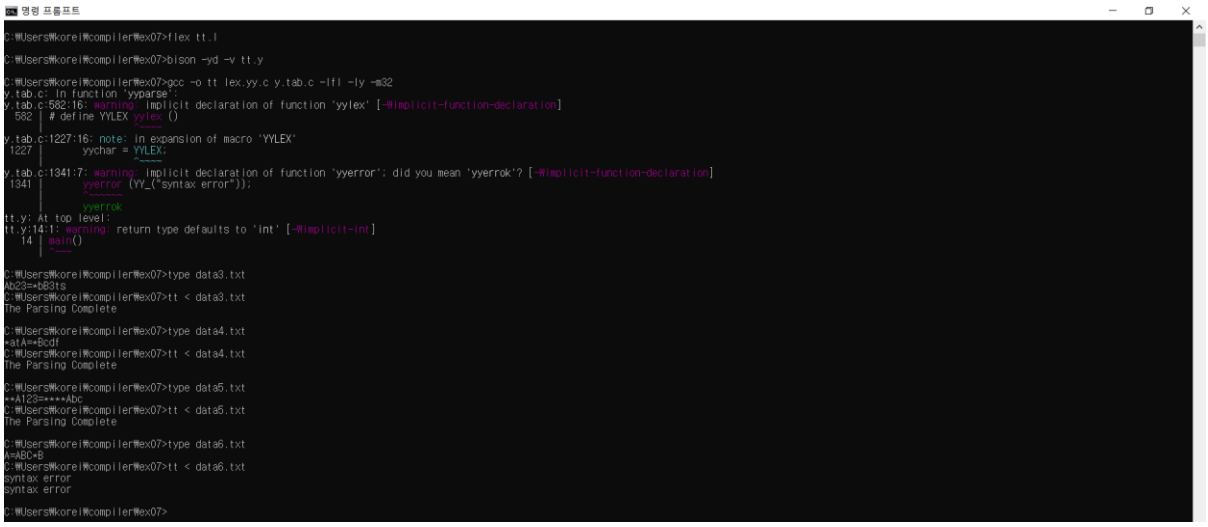
필요한 플렉스 입력 파일(tt.l)은 다음과 같다.

```
%{
#include "y.tab.h"
%}

letter  [A-Za-z]
digit   [0-9]
id      {letter}({letter}|{digit})*

%% /*토큰 정의 규칙*/
"="                 return(yytext[0]);
"*"                 return(yytext[0]);
{id}                return(ID);
[ \n\t\b];
%%
```

바이슨 입력 파일(tt.y)은 다음과 같다.

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

%token ID

%%
S:L '=' R
 |R;
L:'*'R
 |ID;
R:L;
%%
main()
{
    if(yyparse()==0)
    {
        printf("The Parsing Complete \n");
    }
    else
    {
        printf("syntax error \n");
    }
}
```

실행 과정 및 결과는 다음과 같다.



data3.txt : Ab23=*bB3ts
data4.txt : *atA=*Bcdf
data5.txt : **A123=****Abc
data6.txt : A=ABC*B

이 중에서 data6.txt를 제외한 입력은 모두 parsing에 성공하였다. data6.txt의 입력이 문법에 맞지 않음을 알 수 있다.

bison 파일을 실행하면 y.output 파일이 생성되는데 이는 GOTO 그래프를 나타낸다. 이를 이용해 파싱표의 형태로 나타내면 다음과 같다.

상태	구문 분석기 행동				GOTO 함수		
	=	*	id	$	S	R	L
0		s2	s1		3	5	4
1	r4			r4			
2		s2	s1			7	6
3				s8			
4	s9, r5			r5			
5				r2			
6				r5			
7				r3			
8				acc			
9		s2	s1			10	6
10	r1			r1			

파싱표는 충돌이 존재하지만 충돌을 해결하면 s9가 된다. 충돌을 제거하면 예제 6-30에

서 작성한 LALR과 같은 입력에 대해 같은 결과를 출력하는 것을 알 수 있다.

예제 13-8

이론적 내용

예제 13-7과 비슷한 방법으로 플렉스 파일과 바이슨 파일을 작성하면 된다.

소스 코드

필요한 플렉스 입력 파일(tt.l)은 다음과 같다.

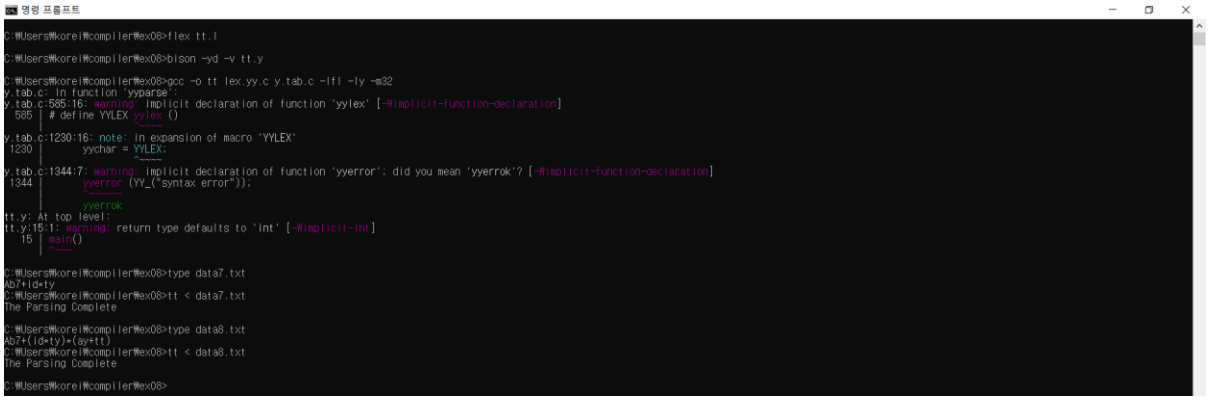
```
%{
#include "y.tab.h"
%}

letter  [A-Za-z]
digit   [0-9]
id      {letter}({letter}|{digit})*

%%/* 토큰 정의 규칙 */
"("                     return(yytext[0]);
"*"                     return(yytext[0]);
")"                     return(yytext[0]);
"+"                     return(yytext[0]);
{id}                    return(ID);

[ \n\t\b]                ;
%%
```

바이슨 입력 파일(tt.y)은 다음과 같다.

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

%token ID

%%
E:E'+'T
|T;
T:T'*'F
|F;
F:'('E')'
```

```
|ID;
%%
main()
{
    if(yyparse()==0)
    {
        printf("The Parsing Complete \n");
    }
    else
    {
        printf("syntax error \n");
    }
}
```

실행 결과와 파싱에 사용된 파싱 표는 다음과 같다.



data7.txt : Ab7+id*ty

data8.txt : Ab7+(id*ty)*(ay+tt)

두 입력 모두 정상적으로 파싱이 되는 것을 알 수 있다.

상태	구문 분석기 행동						GOTO 함수		
	id	+	*	(	)	$	E	T	F
0	s1			s2			3	4	5
1		r6	r6		r6	r6			
2	s1			s2			6	4	5
3		s8				s7			
4		r2	s9		r2	r2			
5		r4	r4		r4	r4			
6		s8			s10				
7						acc			
8	s1			s2				11	5

9	s1			s2					12
10		r5	r5		r5	r5			
11		r1	s9		r1	r1			
12		r3	r3		r3	r3			

마찬가지로 위의 파싱표는 예제 6-26에서 작성한 파싱표와 같은 입력에 대해 같은 결과를 출력한다.

예제 13-9

이론적 내용

예제 13-7과 비슷한 방법으로 플렉스 파일과 바이슨 파일을 작성하면 된다.

소스 코드

필요한 플렉스 입력 파일(tt.l)은 다음과 같다.

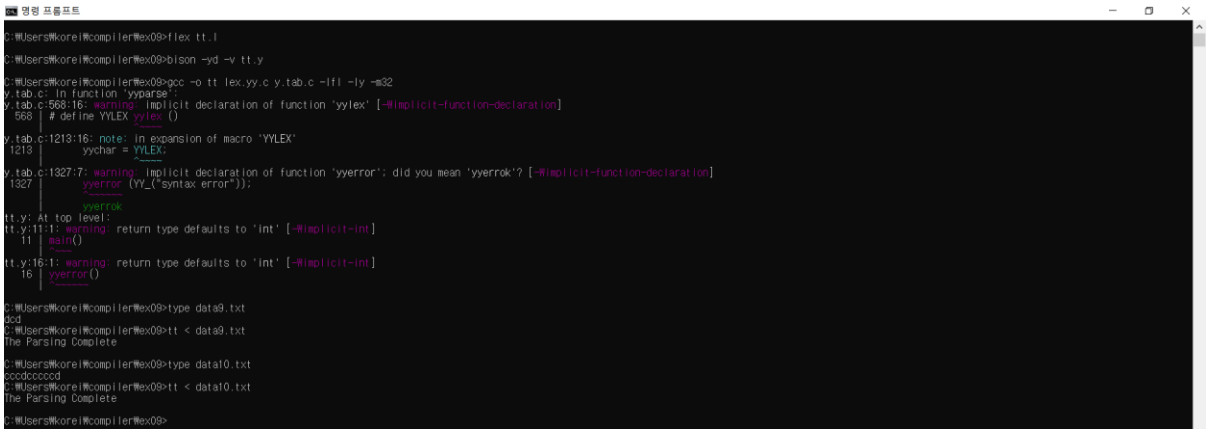
```
%{
%}

%%/* 토큰 정의 규칙 */
"c"                     return(yytext[0]);
"d"                     return(yytext[0]);
[ \n\t\b]                ;
%%
```

바이슨 입력 파일(tt.y)은 다음과 같다.

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

%%
S:C C;
C :'c'C
 | 'd';
%%
main()
{
    yyparse();
    printf("The Parsing Complete\n");
```

```
}
yyerror()
{
    printf("syntax error\n");
    exit(0);
}
```

실행 결과와 파싱에 사용된 파싱 표는 다음과 같다.



data9.txt : dcd

data10.txt : cccdccccd

두 입력 모두 정상적으로 파싱이 되는 것을 알 수 있다.

상태	구문 분석기의 행동			GOTO 함수	
	c	d	$	S	C
0	s1	s2		3	4
1	s1	s2			5
2	r3	r3	r3		
3			s6		
4	s1	s2			7
5	r2	r2	r2		
6			acc		
7	r1	r1	r1		

해당 파싱표는 예제 6-34에서 작성한 파싱표와 결과적으로 같음을 알 수 있다.

예제 13-10

이론적 내용

중위 수식 표기식 방법을 후위 수식 표기식 방법으로 바꾸는 변환이다. 이를 위해 in2po.tab.h 헤더 파일을 포함해야 한다.이를 플렉스 파일의 정의 부분에 기술한다. 수식의 경우 연산자가 루트 노드(부분 트리의 경우 부분 트리의 루트 노드)에, 피연산자가 단말 노드에 존재하는 트리로 나타낼 수 있다. 중위 수식 표기는 이러한 트리를 중위 순회한 결과이고 후위 수식 표기는 후위 순회한 결과이다.

소스 코드

필요한 플렉스 파일 in2po.l은 다음과 같다.

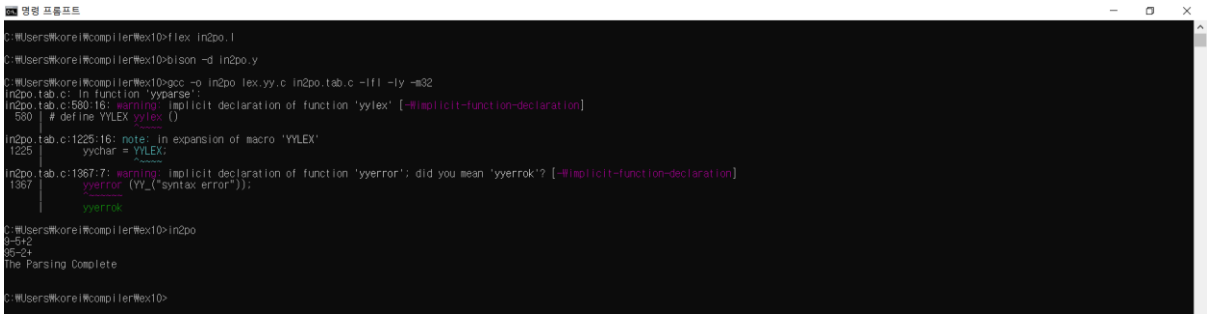
```
%{
#include "in2po.tab.h"
%}

%%

[ \t]+      ;
[0-9]       {yylval = yytext[0] - '0'; return DIGIT;}
[+\-\n]     return yytext[0];

%%
```

필요한 바이슨 파일 in2po.y는 다음과 같다.

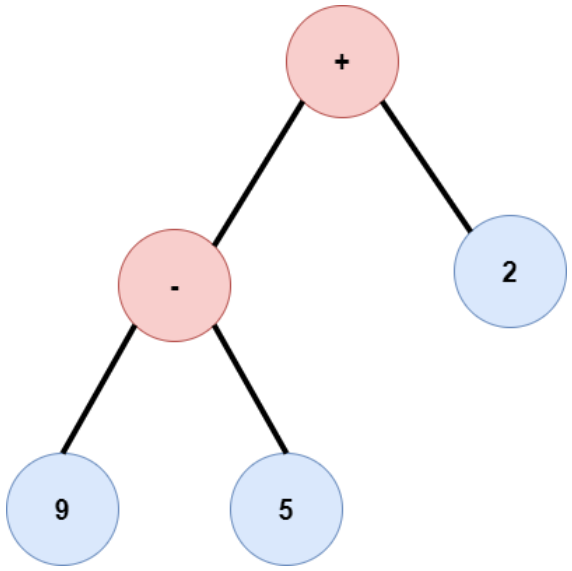
```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token DIGIT

%%

line : expr '\n' {putchar('\n');}
     ;
expr : expr '+' term {putchar('+');}
     | expr '-' term {putchar('-');}
     | term
     ;
term : DIGIT {printf("%d", yylval);}
     ;
```

```
%%

int main()
{
    if(yyparse() == 0) printf("The Parsing Complete\n\n");
    else printf("syntax error \n\n");
}
```

실행 과정 및 결과는 다음과 같다. 실행 과정에서 bison 파일을 생성할 때 in2po.tab.h를 생성하기 위해서는 -d 플래그를 추가해줘야 한다.



9-5+2 수식을 트리로 나타내면 다음과 같다.



이를 중위 순회하면 입력인 9-5+2가 되고 후위 순회하면 출력 결과인 95-2+가 된다.