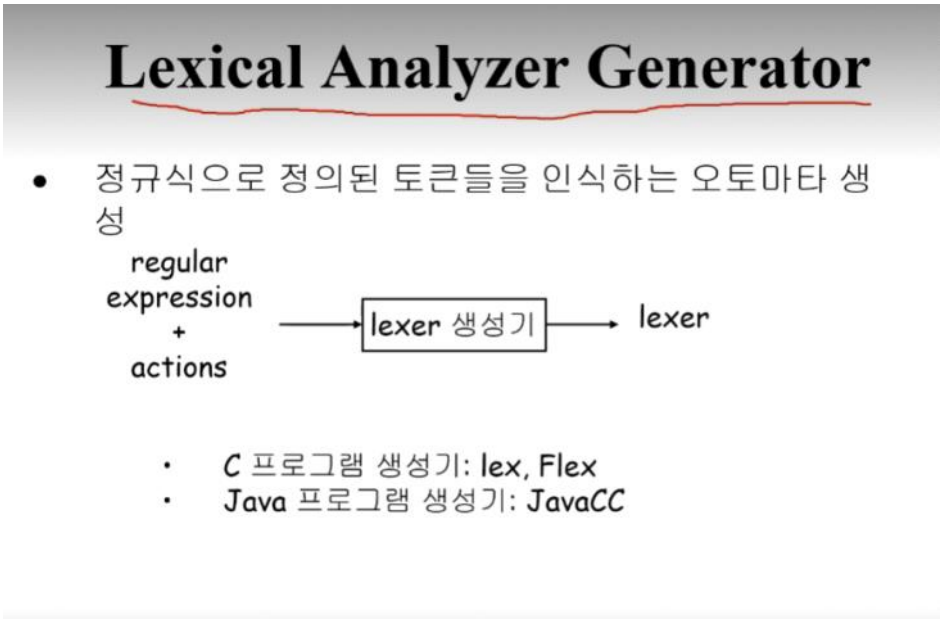
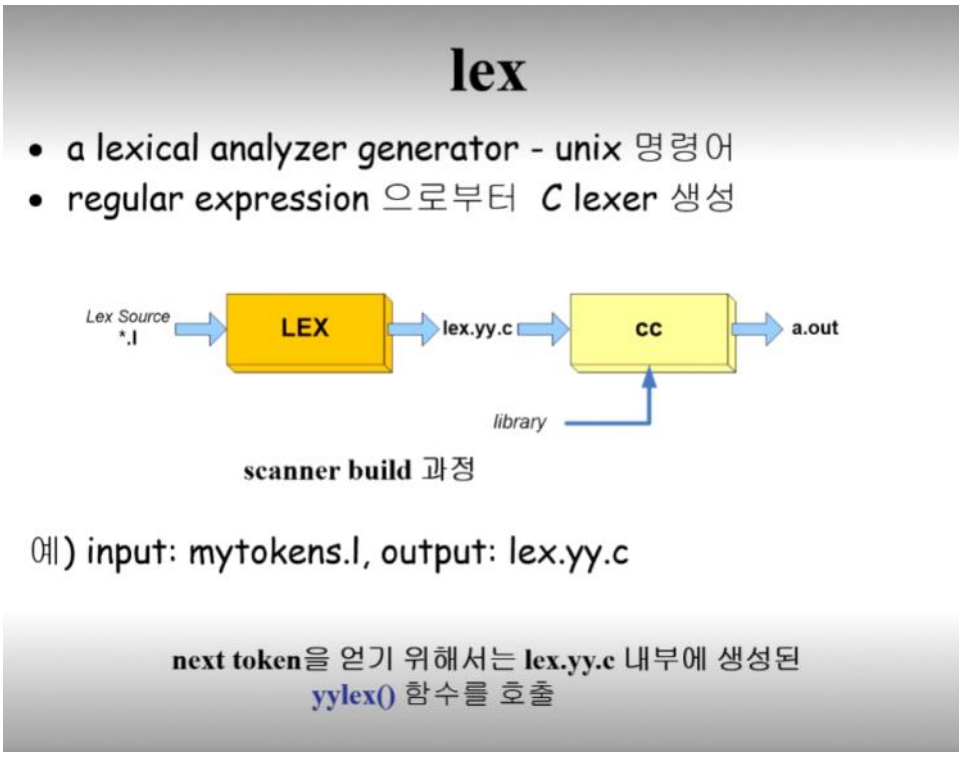
LEX Yacc

2022년 6월 4일 토요일 오후 3:22



화면 캡처: 2022-06-06 오후 8:45



화면 캡처: 2022-06-06 오후 8:49

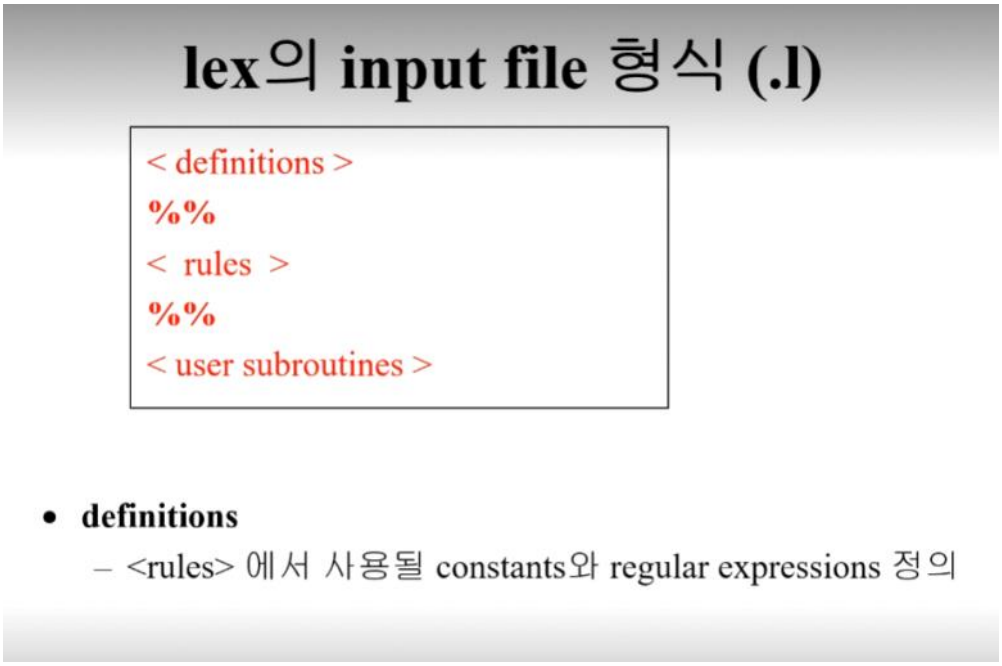
Lex는 unix standard 명령어이다

.l 파일이 입력으로 들어간다 C lexer 생성이 된다 [lex.yy.c] => compile 하면 a.out파일이 생성이 된다

% lex filename.l 명령어 실행 가능

Lex안에 lexer를 호출하고 싶으면 yylex() 함수를 호출한다

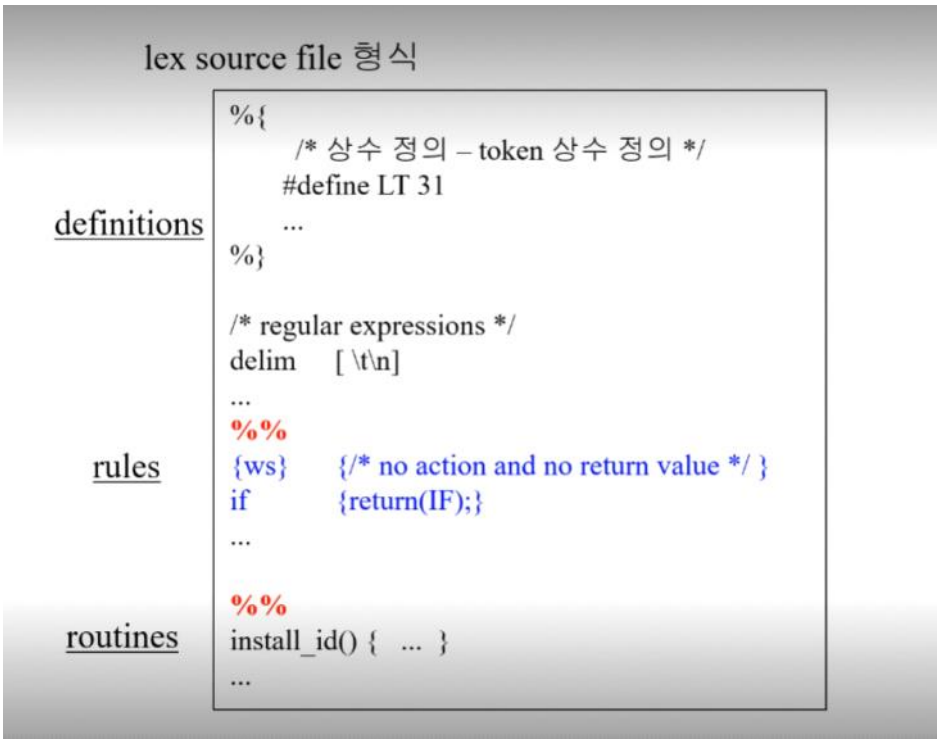
Next Token필요할때 사용



파일 형식 .l

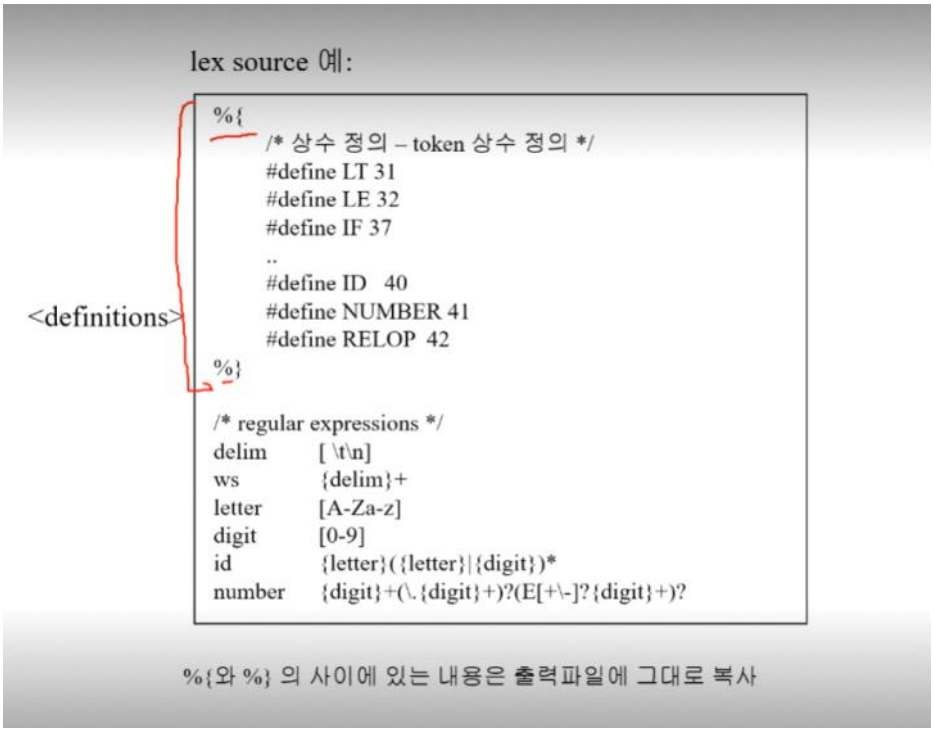
Definitions
<rules>에서 사용될 constants와 regular expressions정의

Lex source file 형식



화면 캡처: 2022-06-06 오후 9:25

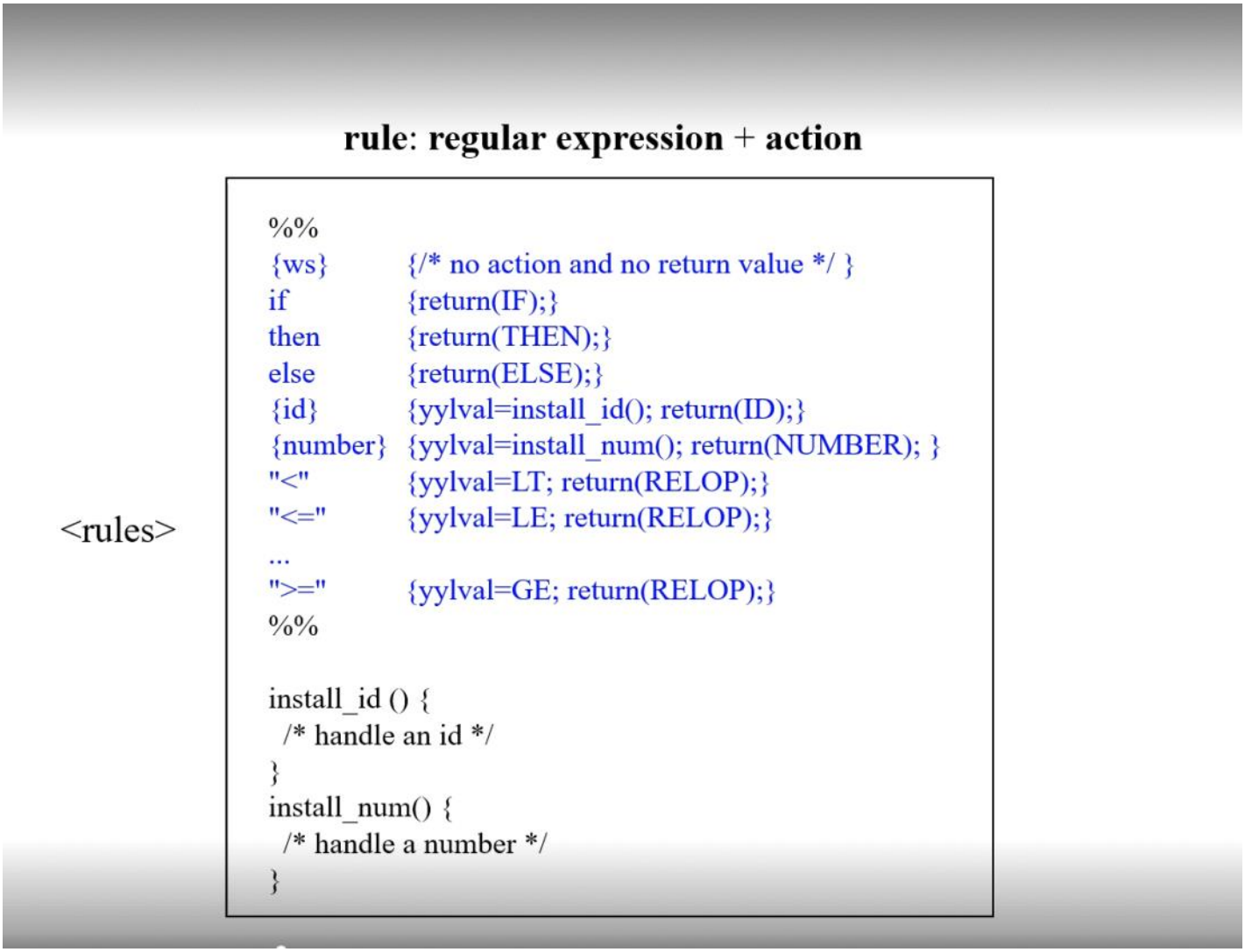
예시



화면 캡처: 2022-06-06 오후 9:28

+는 dagger이다 최소 한번 이상 반복

원래 .는 임의의 문자를 의미하는데 ₩.는 진짜 .를 의미



화면 캡처: 2022-06-06 오후 9:34

Lex actions

정규식에 대응되느 ㄴaction이 있으면 그 액션이 복사 된다

If ----- {return (IF);}

Null action - ignore the input
예 "blank , tab, newline을 무시하라"

predefined names

- yylex() : lexer를 구현한 함수 - token number를 반환
- yytext : token의 실제 문자열
- yylval : token value를 저장한 변수

화면 캡처: 2022-06-06 오후 9:52

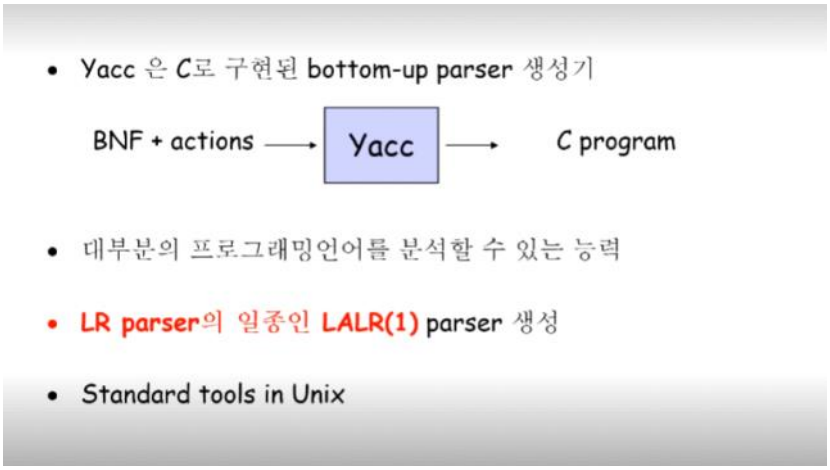
lex Summary

x	the character "x"
x*	0,1,2, ... instances of x.
x | y	an x or y.
xy	x followed by y.
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y
[x-z]	the characters x, y, or z.
[^x]	any character but x.
x?	an optional x.
x+	1,2,3, ... instances of x.
.	any character but newline.

화면 캡처: 2022-06-06 오후 9:55

Yacc

C로 구현된 bottom up parser 생성기 이다

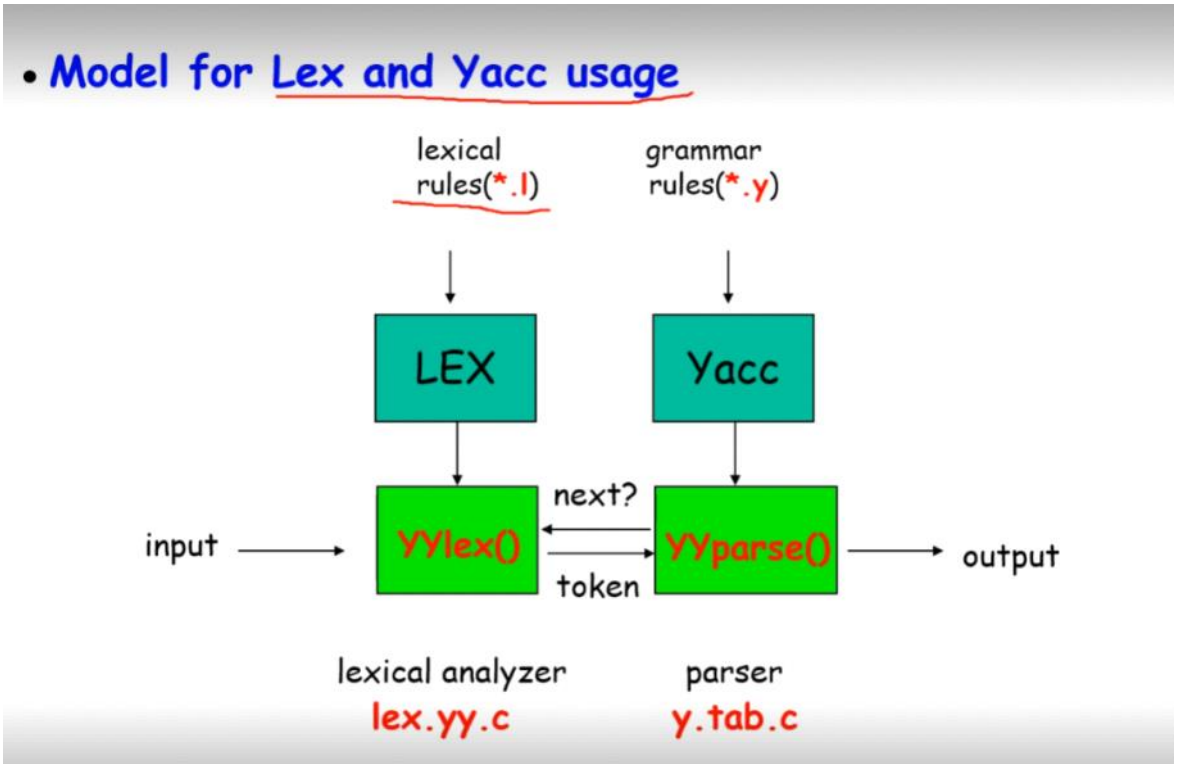


화면 캡처: 2022-06-06 오후 9:59

CFG 가 BNF로 들어간다
LALR(1) parser 생성
EBNF는 못들어간다

JAVA CC는 LL parser 생성
-> top down parser

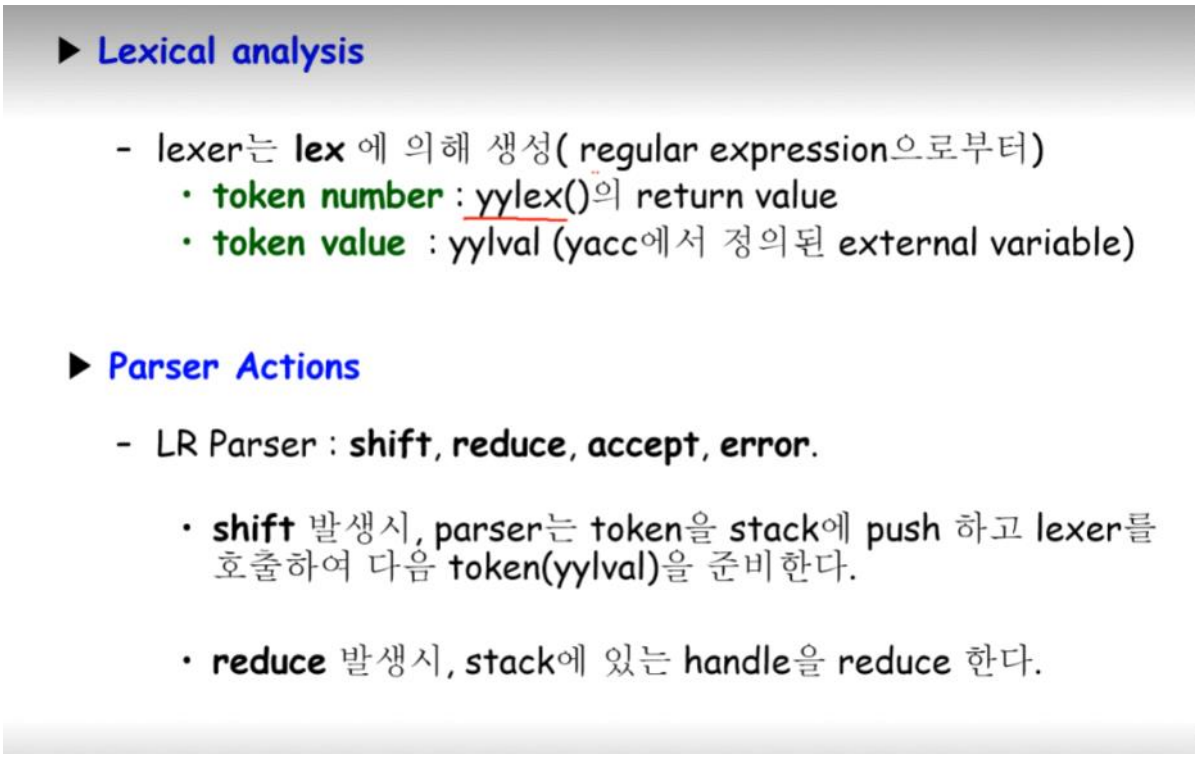
-> EBNF 사용가능
LL(1)을 넘는 부분은 LL(k)로 처리



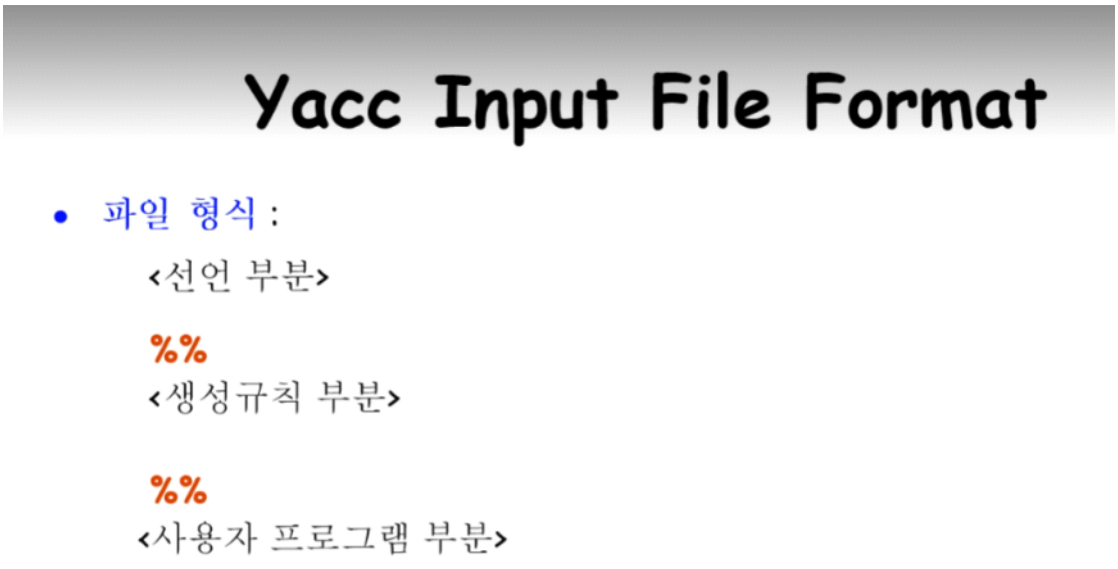
화면 캡처: 2022-06-06 오후 10:19

Lex를 통해 yylex()가 만들어짐 -> lexer

Yac을 통해 yyparse()만들어짐 -> parser



화면 캡처: 2022-06-06 오후 10:23



화면 캡처: 2022-06-06 오후 10:29

- 선언부분
  - 생략 가능(optional)
  - %token token 들의 이름을 선언.
  - 예) %token name1 name2 ...

    => y.tab.h 파일에 token 상수 정의를 자동 생성.
    => lex source에서 y.tab.h 파일을 include 해서 사용.

- 생성규칙부

  grammar rule + action

- 사용자 프로그램 부분
  - 사용자가 입력한 코드를 그대로 출력 프로그램에 복사

화면 캡처: 2022-06-06 오후 10:30

%token keyword쓰면 자동으로 순서 정해준다

y.tab.h 파일에 token 상수 정의를 자동 생성
Lex source 에서 y.tab.h파일을 include해서 사용가능

생성규칙부분

- production + action

action : 해당 문장에 대해 수행할 C statements.

예)

```
expression : expression '+' term { printf("addition detected\n"); }
```

생성규칙부분

- **production** 표기 형식

CFG

expression -> expression + term | term

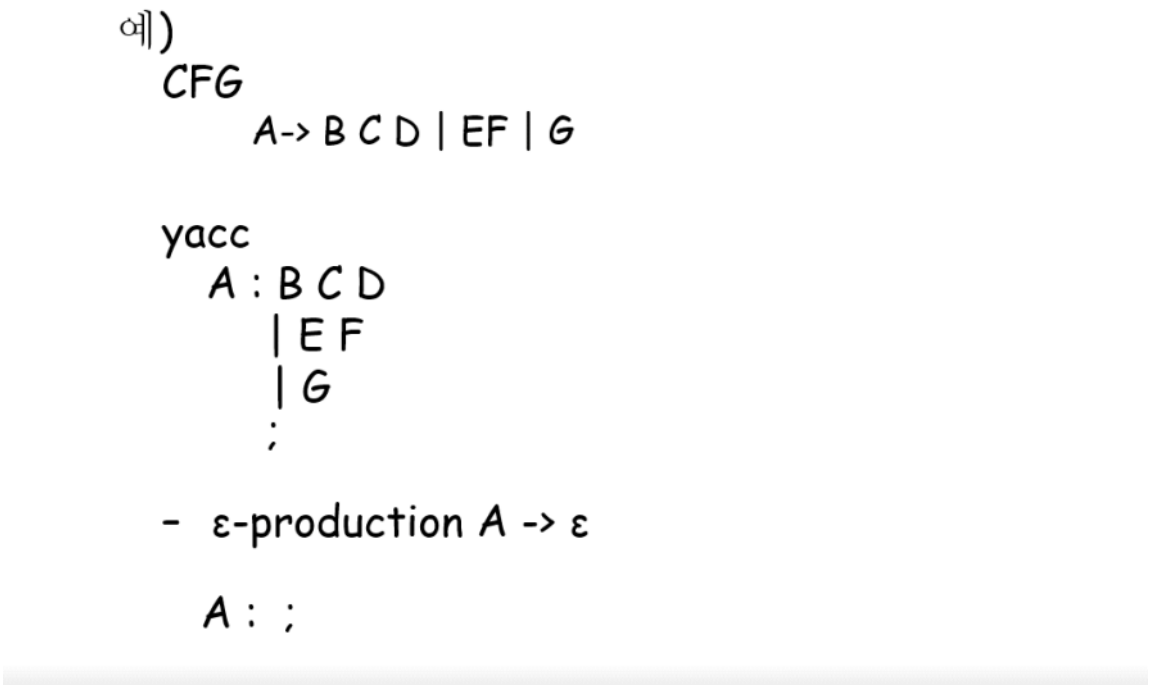
yacc

```
expression  :  expression '+' term  |  term
            ;
```

- 각 **rule**은 **; (semicolon)**으로 종료
- 한 글자 **token**은 **single quotes** 안에 넣어 직접 표시 가능
  '+', '-'

화면 캡처: 2022-06-06 오후 10:40

생성규칙 예



화면 캡처: 2022-06-06 오후 10:42

->는 :
끝에는 ;으로 마무리 한다

Action 정의 형식

- 임의의 **C statements**를 **{ }** 안에 정의.

예)

```
expression : expression '+' term { printf("addition detected\n"); }
                   ;
expression : term    {printf("simple expression\n"); }
                   ;
```

- 실제 파서 구현시에는 action으로 파스트리 생성을 위한 코드를 정의할 수 있음.

- action을 규칙 중간에 삽입할 수도 있다.

화면 캡처: 2022-06-06 오후 10:43

실제 파서 구현시에는 action으로 파스 트리 생성을 위한 코드를 정의할수있다

중간에도 action 삽입 가능

Action 정의 형식

- 생성규칙에 있는 symbol 의 attribute 값은 pseudo variable $n 형식으로 지정
  - **$$** : 좌변 **nonterm**의 속성 값
  - **$1**, **$2**, … 우변 1번째, 2번째 symbol의 속성 값.

- 예: 수식의 계산 수행

```
expr : expr '+' expr  { $$ = $1 + $3; }
```

- 예: Parse tree construction
  - **node**('+',n1,n2) 를 n1, n2를 자식으로 가진 '+' subtree를 반환하는 함수라고 하자.

```
ex) expr : expr '+' expr  { $$ = node('+',$1,$3); }
```

화면 캡처: 2022-06-06 오후 10:44

```
expr : expr '+' expr {$$ = $1 + $3;}
```

$$은 nonterm이다

$1의 첫번째 expr의미 $3은 3번째 expr을 의미이다

$$ = $1 + $3; 을 nod('+',$1,$3}으로 정의할수도 있다

Ambiguity and Conflicts

- Yacc은 shift-reduce parser (LR parser)
  - Parsing actions: shift, reduce, accept, error.
- Ambiguity
  - ambiguity 가 있으면 parsing action에서 충돌 발생
- Conflicts
  shift/reduce, reduce/reduce
  - Yacc 은 두가지 disambiguating rules 사용
  - shift/reduce conflict 발생하면 shift를 수행
  - reduce/reduce conflict 경우, 먼저 선언된 생성규칙을 적용
- 사용자가 Ambiguity 제거 규칙을 정의하면 모호성을 가진 문법도 분석 가능

화면 캡처: 2022-06-06 오후 10:49

Yacc은 LR parser

모호성이 있으면 parsing conflict 발생

Conflict 처리 방법 명시 되어있다

Yacc에서 conflict처리 어떻게 하는지 위에 내용에 있다

Precedence and Associativity

- %left, %right : associativity 선언

ex)

```
%right '='
%left '+' '-'
%left '*' '/'
%%
expr : expr '=' expr
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     ;
```

이 경우,

a = b = c * d - e - f * g 은 a = (b = (((c * d) - e) - (f * g))) 으로 처리

화면 캡처: 2022-06-06 오후 10:55

Right는 우결합 부터

정수 연산을 지원하는 계산기

- **lex source - calc.l**

```
%{
/* LEX source for calculator program */
#include "y.tab.h"          // a header file to be generated by yacc
extern int yylval;          // token value
%}
%%
[ \t]    ;          /* ignore blanks and tabs */
[0-9]+              {yylval := atoi(yytext); return NUMBER;}
"mod"               return MOD;
"div"               return DIV;
"sqr"               return SQR;
...
 %%
```

화면 캡처: 2022-06-06 오후 10:59

- calc.y

```
%{
/* YACC source for calculator program */
# include <stdio.h>
%}
%token NUMBER DIV MOD SQR
%left '+' '-'
%left '*' DIV MOD
%left  SQR
%%
arithmeticExpression : expr {
            printf("%d\n", $1);
             return 0;
          }
      ;

expr : '(' expr ')'  {$$ = $2;}
   | expr '+' expr {$$ = $1 + $3;}
   | expr '-' expr {$$ = $1 - $3;}
   | expr '*' expr {$$ = $1 * $3;}
   | expr MOD expr {$$ = $1 % $3;}
   | SQR  expr    {$$ = $2 * $2;}
   | NUMBER      {$$ = $1;}
```

화면 캡처: 2022-06-06 오후 11:00