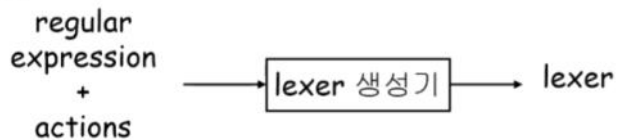


Lexical Analyzer Generator

- 정규식으로 정의된 토큰들을 인식하는 오토마타 생성

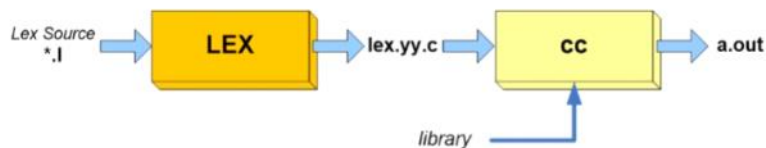


- C 프로그램 생성기: **lex**, **Flex**
- Java 프로그램 생성기: **JavaCC**

화면 캡처: 2022-06-06 오후 8:45

lex

- a lexical analyzer generator - unix 명령어
- regular expression 으로부터 C lexer 생성



scanner build 과정

예) input: mytokens.l, output: lex.yy.c

next token을 얻기 위해서는 lex.yy.c 내부에 생성된
yylex() 함수를 호출

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

lex의 input file 형식 (.l)

```
< definitions >
%%
< rules >
%%
< user subroutines >
```

- **definitions**

- <rules> 에서 사용될 constants와 regular expressions 정의

파일 형식 .l

Definitions

<rules>에서 사용될 constants와 regular expressions정의

Lex source file 형식

lex source file 형식

definitions

rules

routines

```
%{
    /* 상수 정의 - token 상수 정의 */
    #define LT 31
    ...
}%

/* regular expressions */
delim [ \t\n]
...

%%
{ws}    { /* no action and no return value */ }
if      {return(IF);}
...

%%
install_id() { ... }
...
```

예시

lex source 예:

<definitions>

```
%{  
/* 상수 정의 - token 상수 정의 */  
#define LT 31  
#define LE 32  
#define IF 37  
..  
#define ID 40  
#define NUMBER 41  
#define RELOP 42  
%}  
  
/* regular expressions */  
delim    [ \t\n]  
ws       {delim}+  
letter   [A-Za-z]  
digit    [0-9]  
id       {letter}({letter}|{digit})*  
number   {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
```

%{와 %} 의 사이에 있는 내용은 출력파일에 그대로 복사

+는 dagger이다 최소 한번 이상 반복

원래 .는 임의의 문자를 의미하는데 \.는 진짜 .를 의미

rule: regular expression + action

<rules>

```
%%  
{ws}      { /* no action and no return value */ }  
if         {return(IF);}   
then       {return(THEN);}   
else       {return(ELSE);}   
{id}       {yylval=install_id(); return(ID);}   
{number}   {yylval=install_num(); return(NUMBER); }   
"<"        {yylval=LT; return(RELOP);}   
"<="       {yylval=LE; return(RELOP);}   
...   
">="       {yylval=GE; return(RELOP);}   
%%  
  
install_id () {  
    /* handle an id */  
}  
install_num() {  
    /* handle a number */  
}
```

화면 캡처: 2022-06-06 오후 9:34

Lex actions

정규식에 대응되는 action이 있으면 그 액션이 복사 된다

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

Null action - ignore the input

예 "blank , tab, newline을 무시하라"

predefined names

- `yylex()` : `lexer`를 구현한 함수 - `token number`를 반환
- `yytext` : `token`의 실제 문자열
- `yyval` : `token value`를 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

- Yacc 은 C로 구현된 bottom-up parser 생성기



- 대부분의 프로그래밍언어를 분석할 수 있는 능력
- LR parser의 일종인 LALR(1) parser 생성
- Standard tools in Unix

화면 캡처: 2022-06-06 오후 9:59

CFG 가 BNF로 들어간다

LALR(1) parser 생성

EBNF는 못들어간다

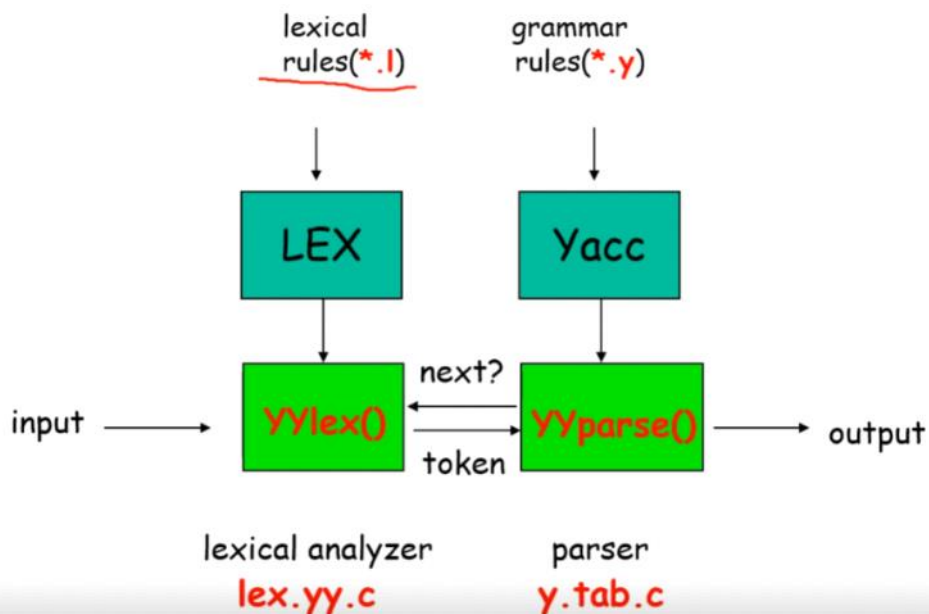
JAVA CC는 LL parser 생성

-> top down parser

-> EBNF 사용가능

LL(1)을 넘는 부분은 LL(k)로 처리

• Model for Lex and Yacc usage



화면 캡처: 2022-06-06 오후 10:19

Lex를 통해 yylex()가 만들어짐 -> lexer

Yac을 통해 yyparse()만들어짐 -> parser

► Lexical analysis

- lexer는 lex 에 의해 생성(regular expression으로부터)
 - **token number** : yylex()의 return value
 - **token value** : yylval (yacc에서 정의된 external variable)

► Parser Actions

- LR Parser : **shift, reduce, accept, error.**
 - **shift** 발생시, parser는 token을 stack에 push 하고 lexer를 호출하여 다음 token(yylval)을 준비한다.
 - **reduce** 발생시, stack에 있는 handle을 reduce 한다.

화면 캡처: 2022-06-06 오후 10:23

Yacc Input File Format

• 파일 형식 :

<선언 부분>

%%

<생성규칙 부분>

%%

<사용자 프로그램 부분>

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 \rightarrow 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
'+', '-'

생성규칙 예

예)

CFG

$A \rightarrow B C D \mid E F \mid G$

yacc

$A : B C D$

$\mid E F$

$\mid G$

;

- ϵ -production $A \rightarrow \epsilon$

$A : ;$

화면 캡처: 2022-06-06 오후 10:42

->는 :

끝에는 ;으로 마무리 한다

Action 정의 형식

- 임의의 C statements를 { } 안에 정의.

예)

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

```
expression : term {printf("simple expression\n"); }  
            ;
```

- 실제 파서 구현시에는 **action**으로 파스 트리 생성을 위한 코드를 정의할 수 있음.
- **action**을 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 nonterm의 속성 값
 - $\$1, \$2, \dots$ 우변 1번째, 2번째 symbol의 속성 값.
- 예: 수식의 계산 수행
 $\text{expr} : \text{expr} '+' \text{expr} \{ \$\$ = \$1 + \$3; \}$
- 예: Parse tree construction
 - $\text{node}('+', n1, n2)$ 를 $n1, n2$ 를 자식으로 가진 '+' subtree를 반환하는 함수라고 하자.

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

화면 캡처: 2022-06-06 오후 10:44

$\text{expr} : \text{expr} '+' \text{expr} \{ \$\$ = \$1 + \$3; \}$

$\$ \$$ 은 nonterm이다

$\$1$ 의 첫번째 expr 의미 $\$3$ 은 3번째 expr 을 의미이다

$\$ \$ = \$1 + \$3;$ 을 $\text{nod}('+', \$1, \$3)$ 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;
...
%%
```

- 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;}
```