Yacc & Bison

COMP321 컴파일러

2007년 가을학기

경북대학교 전자전기컴퓨터학부

Lex and Yacc

- Lex and Yacc are
  tools designed for compiler and interpreter writers
  but they can also be useful for many other applications which handle structured input.
- They allow for
  - rapid prototyping
  - easy modification (of input structure)
  - simple maintenance
- GNU variants
  - **flex** ← lex
  - **bison** ← yacc

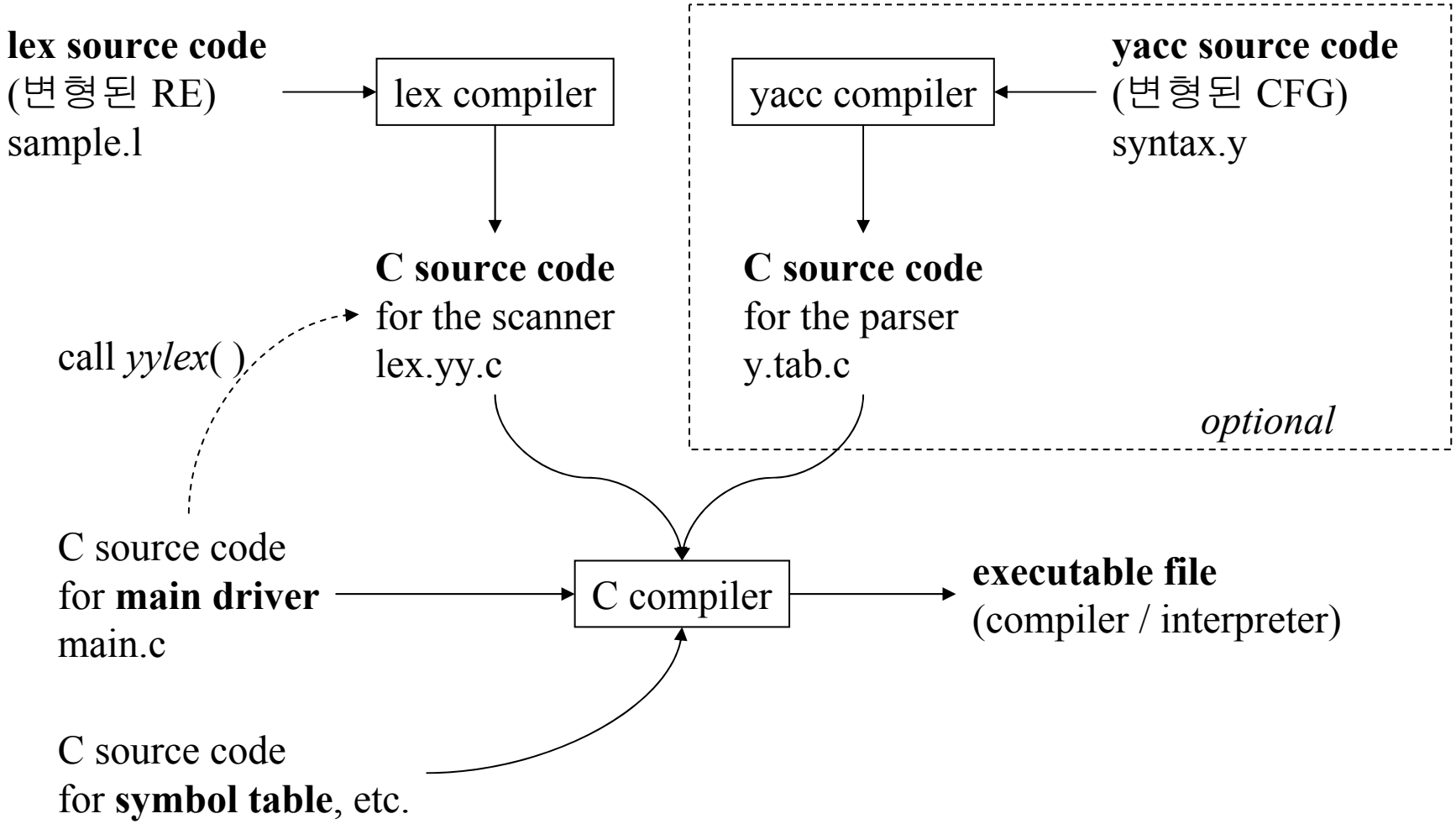
Lex and Yacc

- **Sample applications** include
  - the desktop calculator, ***bc***
  - the typesetting preprocessor tools, ***eqn*** and ***pic***
  - the portable C compiler ***pcc***
  - the GNU C compiler ***gcc***
  - a **SQL** syntax checker
  - the ***lex*** program itself

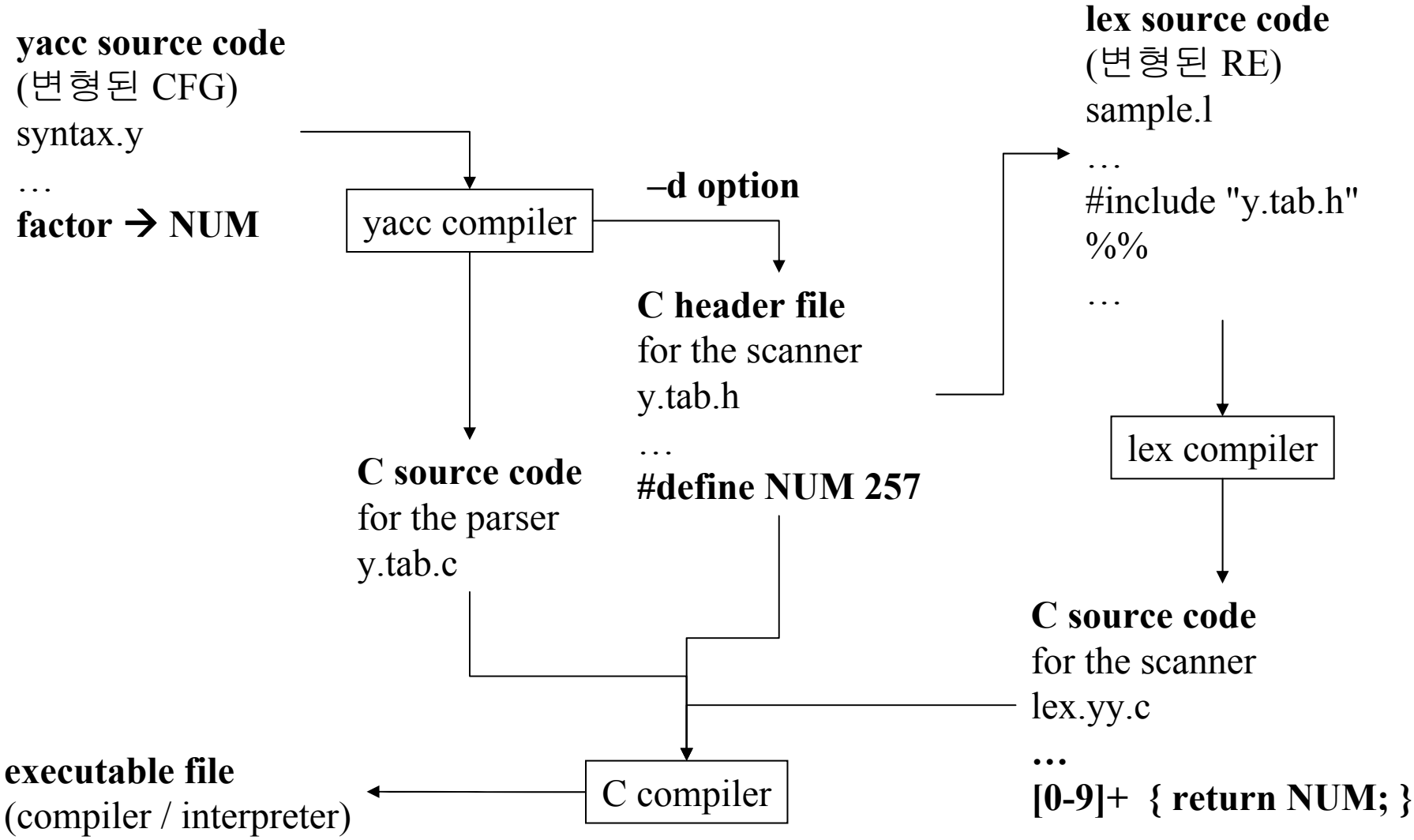
Yacc

- Yet Another Compiler-Compiler
  - compiler를 compile하는 program
- S. C. Johnson and R. Sethi,
  "Yacc: A parser generator",
  Unix Research System Programmer's Manual,
  Tenth Edition, Volume 2
- **LALR(1) parser**
  - grammar rule → C source code
  - not as fast as hand-written parser
  - but, rapid prototyping & easy for maintenance
  - easy to cooperate with lex

Global View



More Practical View



Yacc Source Code

- file structure

  declarations

  %%

  rules

  %%

  programs

- similar to lex file structure !

Declarations Part

- **%{**
  …
  %}
  - y.tab.c 로 그대로 copy (lex에서와 동일)
- **%token NAME1 NAME2 …**
  - CFG에서 사용될 terminal을 미리 정의
  - %token으로 정의되지 않으면, non-terminal 로 가정
  - 단, single quoted character는 declare 하지 않아도 terminal로 취급: '+', '–', '=', …
- **%start symbol**
  - goal non-terminal 을 선언
  - 선언하지 않으면, rule 중에서 제일 처음 것 사용

Rules Part

- 변형된 CFG grammar

```
stmt:       IDEN '=' expr
      |     expr
      ;
expr:       expr '+' term
      |     term
      ;
```

  - : (colon) 기준으로 LHS → RHS를 의미
  - | (bar) 는 이전의 LHS를 그대로 사용함을 의미
  - ; (semi-colon) 은 rule의 끝

Rules Part

- 변형된 CFG grammar

```
expr:     expr + term          { $$ = $1 + $3; }
    |     term                 { $$ = $1; }
    ;
```

- action 부분
  - { } 로 묶어서, rule 의 뒤에 첨가 가능 (줄 바꿈 가능)
  - 모든 C 언어 사용 가능
  - 생략 시, do nothing (only syntax check)
- $$ 변수들
  - **모든 yacc symbol은 value를 가짐 (int type)**
  - **$$** : LHS 의 value
  - **$1, $2, $3, …** : RHS #1, #2, #3, … 의 value

Programs Part

- main( ) function !

```
int main(void) {
  return yyparse( );
}
```

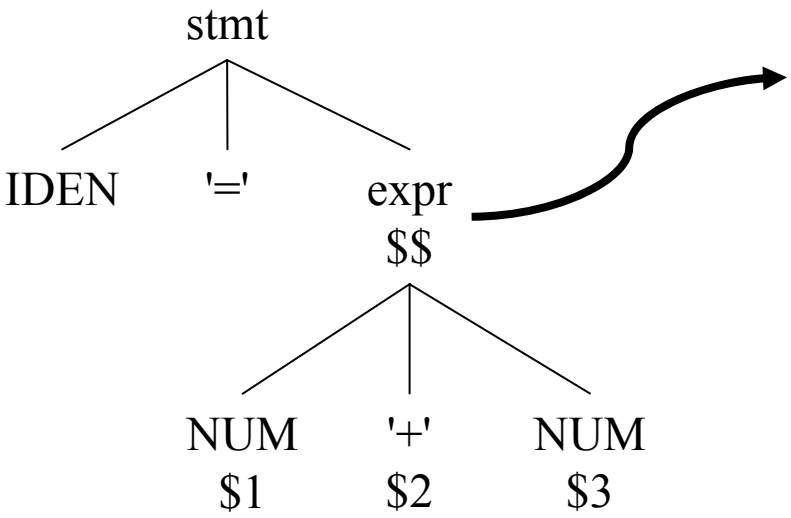
- and any needed C functions
  - 보통, yyerror 를 자주 선언

```
int yyerror(char* str) {
  return fprintf(stderr, str);
}
```

Parse Tree and Reduction

- yacc는 parse tree를 만든다는 개념으로 접근하지만,
  실제 parse tree를 만들지는 않는다.
  - parse tree를 만들려면, action 부분에 code 필요

  expr: NUM + NUM { $$ = $1 + $3; }



expr → NUM + NUM 로
reduce action 일어나는 순간,
action 수행

parse tree를 만들려면?
$$ = MakeTree($1, $2, $3);
(MakeTree 함수는 구현해야 함!)

An Example: sample.y

```
%{
#include <stdio.h>
%}
%token IDEN NUM
%%
stmt:  IDEN '=' expr  { printf("%s = %d\n", $1, $3); }
      |  expr                { printf("%d\n", $1); }
      ;
expr: expr '+' term  { $$ = $1 + $3; }
      | expr '-' term   { $$ = $1 - $3; }
      | term              { $$ = $1; }
      ;
term: term '*' factor { $$ = $1 * $3; }
      |  term '/' factor { if ($3 == 0) yyerror("divide by zero\n");
                           else $$ = $1 / $3; }
      | factor             { $$ = $1; }
      ;
factor: '-' NUM        { $$ = -$2; }
       | NUM             { $$ = $1; }
       | '(' expr ')'       { $$ = $2; }
       ;
```

```
%%

int main(void) {
       int code = yyparse();
       return 0;
}

int yyerror(char* str) {
       return fprintf(stderr, str);
}
```

- compile

```
$ yacc –d sample.y

$ bison –d sample.y
```

An Example: sample.l

- in y.tab.h

```
…
#define IDEN 258
#define NUM 259
…
```

- in sample.l

```
%{
#include "y.tab.h"
extern int yylval; /* defined in yacc */
%}
%%
[0-9]+      { yylval = (int)strtol(yytext, NULL, 10);
              return NUM; }
[a-zA-Z]+       { yylval = (int)yytext;
                  return IDEN; }
[ \t]       ;
\n          return 0;
.           return yytext[0];
```

- compile

```
$ lex sample.l
$ cc y.tab.c lex.yy.c -lfl
```

- 실행 결과

```
$ a.exe
3 + 4 * – 5 + 7        (user input)
–10                    (program output)
$
```

Shift/Reduce Conflicts

- 아래는 LALR(1)이 아닌 grammar

```
%{
#include <stdio.h>
%}
%token IDEN NUM
%%
stmt:       IDEN '=' expr           { printf("%s = %d\n", $1, $3); }
      |     expr                    { printf("%d\n", $1); }
      ;
expr:       expr '+' expr { $$ = $1 + $3; }
      |     expr '-' expr  { $$ = $1 - $3; }
      |     expr '*' expr { $$ = $1 * $3; }
      |     expr '/' expr  { if ($3 == 0) yyerror("divide by zero\n"); else $$ = $1 / $3; }
      |     '-' expr       { $$ = -$2; }
      |     '(' expr ')'   { $$ = $2; }
      |     NUM            { $$ = $1; }
      ;
%%
…
```

Shift/Reduce Conflicts

- yacc를 실행하면,
  - **$ yacc –d sample2.y**
    conflicts: **20 shift/reduce**
  - grammar가 valid 한 지 check 에 사용 가능 !
    - hand-written parse에서도 yacc는 유용하다
- (operator) precedence : shift/reduce conflict 제거의 편법
  - precedence가 낮은 것부터 높은 것 순서로 나열

    %right '='

    %left '+' '–'

    %left '*' '/'

Another Example

- 완전한 (grammar + precedence)

```
%{
#include <stdio.h>
%}
%token IDEN NUM
%left '-' '+'
%left '*' '/'
%%
stmt:       IDEN '=' expr           { printf("%s = %d\n", $1, $3); }
      |     expr                    { printf("%d\n", $1); }
      ;
expr:       expr '+' expr { $$ = $1 + $3; }
      |     expr '-' expr  { $$ = $1 - $3; }
      |     expr '*' expr { $$ = $1 * $3; }
      |     expr '/' expr  { if ($3 == 0) yyerror("divide by zero\n"); else $$ = $1 / $3; }
      |     '-' expr       { $$ = -$2; }  %prec '*'      /* unary minus 처리 ! */
      |     '(' expr ')'    { $$ = $2; }
      |     NUM          { $$ = $1; }
      ;
%%
…
```

실행 결과

$ a.exe
99 + 12
111
$ a.exe
2 + 3 – 14 + 33
24
$ a.exe
100 + –50
50

Bison Extension for Value Type

- yacc의 각 symbol 은 int value를 가질 수 있다.
  - parse tree build 나 다른 작업에는 불편
- **%union {**
  double dval;
  int ival;
  }
- in y.tab.h,
  - **typedef union {**
    double dval;
    int ival;
    } YYSTYPE;
    extern YYSTYPE yylval;

Bison Extension for Value Type

- lex에서는 yylval 이 union type 이 됨
  - … { yylval.dval = **strtod(yytext, NULL);**
    return REAL; }
- yacc rule에서는 $$, $1, $2, … 가 pointer to union
  - … { $$->dval = **$1->dval + $3->dval;** }
- yacc rule에서 항상 어느 field를 쓰게 하는 방법
  - **%token <ival> INT**
  - **%token <dval> REAL**
    - When lex return REAL, $3 means $3→dval
  - **%type <dval> expr**
    - expr non-terminal은 항상 dval 만 사용

Example: yacc file

```
%{
#include <stdio.h>
%}
%union {
        int ival;
        double dval;
}
%token <ival> INT
%token <dval> REAL
%type <dval> stmt expr
%left '-' '+'
%left '*' '/'
%%
stmt: expr                  { printf("%f\n", $1); }
        ;
expr:   expr '+' expr       { $$ = $1 + $3; }
        | expr '-' expr     { $$ = $1 - $3; }
        | expr '*' expr     { $$ = $1 * $3; }
        | expr '/' expr     { $$ = $1 / $3; }
        | '-' expr          { $$ = -$2; }    %prec '*'
        | '(' expr ')'      { $$ = $2; }
        | INT               { $$ = (double)($1); }
        | REAL              { $$ = $1; }
        ;
%%

int main(void) {
        yyparse();
        return 0;
}
```

Example: lex file

```
%{
#include "y.tab.h"
%}
%%
[0-9]+              { yylval.ival = (int)strtol(yytext, NULL, 10);
                      return INT; }
[0-9]+"."[0-9]+     { yylval.dval = strtod(yytext, NULL);
                     return REAL; }
[ \t]               ;
\n                                  return 0;
.                                   return yytext[0];
%%
```

- compile

```
$ bison –d sample.y
$ flex sample.l
$ cc y.tab.c lex.yy.c -lfl
```

- 실행 결과

```
$ a.exe
3 + 4.5 * 1.2
8.400000
```