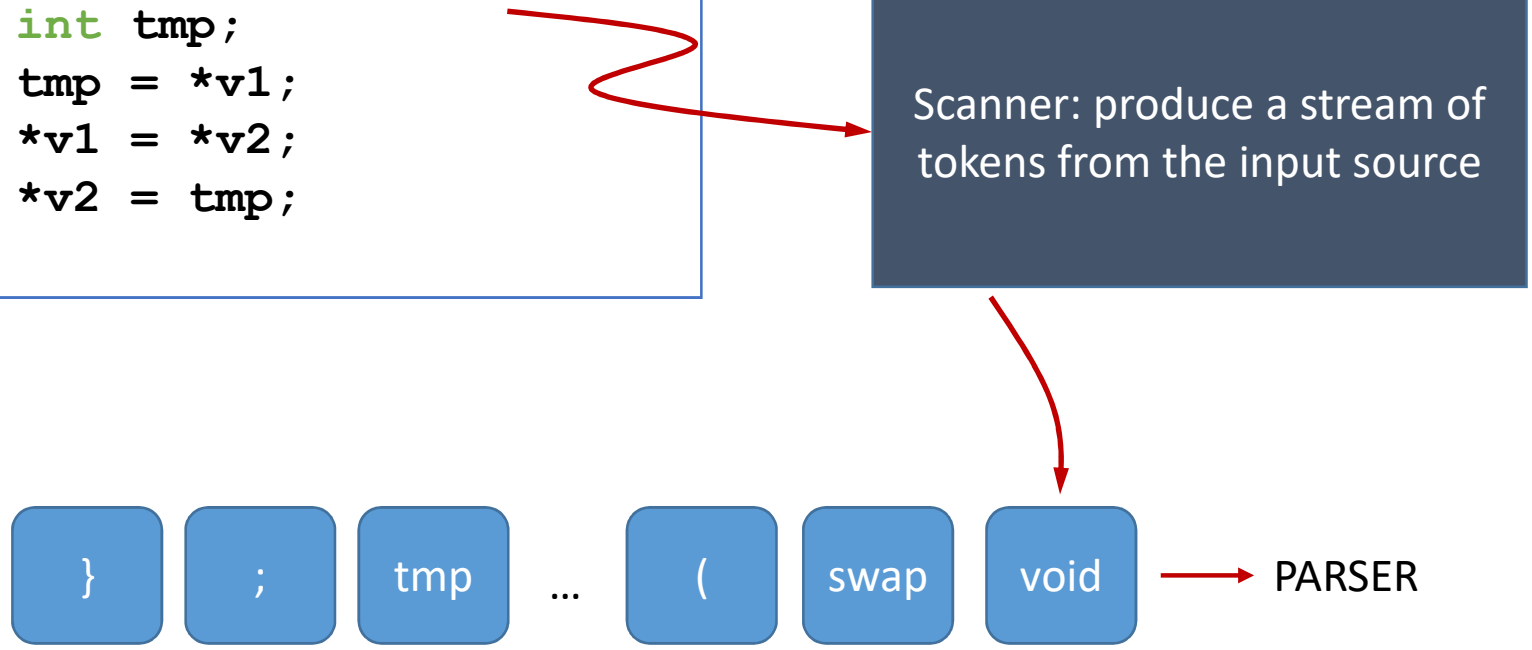
Programming Languages
Lecture04 – Lex and Yacc

남 범 석

bnam@skku.edu

Lexical Analysis

```
void swap (int *v1, int *v2)
{
      int tmp;
      tmp = *v1;
      *v1 = *v2;
      *v2 = tmp;
}
```



lex / flex

- lex is a scanner generator
  - Input is a set of regular expressions and associated actions (written in C)
  - Output is a table-driven scanner (lex.yy.c)
  - GNU flex: an open source implementation of the original UNIX lex utility

lex input

- Structure of lex input

```
DEFINITIONS
%%
pattern      action
…
%%
USER SUBROUTINES
```

- Example

```
digit [0-9]
letter [a-zA-Z]
$ cat ex1.l
%%
"hello world"
{letter}({letter}|{digit})*
.
%%
void myprint(char* text){ … }
```

```
printf("GOODBYE!\n");
myprint(yytext);
;
```

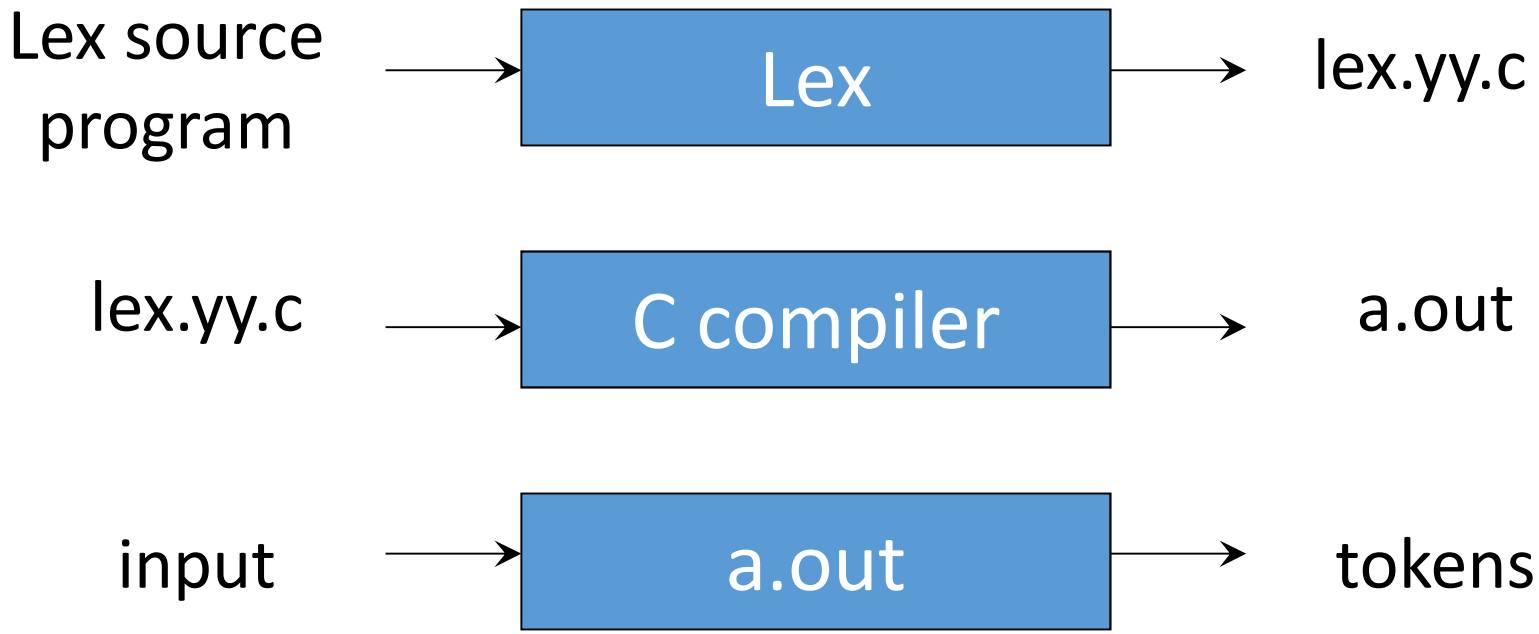
Prints out "GOODBYE!" anytime the string "hello world" is encountered.

Does nothing for any other character.

Using lex

- The table is translated to a C program (lex.yy.c) which
  - reads an input stream
  - partitioning the input into strings which match the given expressions and
  - copying it to an output stream if necessary

An Overview of Lex



Using lex

- Process the lex file to generate a scanner
  - `$ lex ex1.l`
  - The scanner is saved as lex.yy.c

- Compile the scanner and grab main() from the lex library using –ll option
  - `$ cc lex.yy.c -ll`

- Run the scanner taking input from the standard input
  - ```
    $ ./a.out
    hello world
    GOODBYE!
    ```
    ```

lex pattern examples

Metacharacter	Matches
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`?`	zero or one copy of the preceding expression
`^`	beginning of line / complement
`$`	end of line
`a|b`	`a` or `b`
`(ab)+`	one or more copies of `ab` (grouping)
`[ab]`	`a` or `b`
`a{3}`	3 instances of `a`
`"a+b"`	literal "`a+b`" (C escapes still work)

lex pattern examples

Pattern	
abc	Match the string “abc”
[a-zA-Z]	Match any lower or uppercase letter
dog.*cat	Match any string starting with dog, and ending with cat
(ab)+	Match one or more occurrences of “ab” concatenated
[^a-z]+	Matches any string of one or more characters that do not include lower case a-z
[+-]?[0-9]+	Match any string of one or more digits with an optional prefix of + or -
^[a-z]+	Matches any line starting with a lower case letter

Predefined Variables in lex

Name	Function
`char *yytext`	pointer to matched string
`int yyleng`	length of matched string
`FILE *yyin`	input stream pointer
`FILE *yyout`	output stream pointer
`int yylex(void)`	call to invoke lexer, returns token
`char* yymore(void)`	return the next token
`int yyless(int n)`	retain the first n characters in yytext
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`ECHO`	write matched string
`REJECT`	go to the next alternative rule
`INITAL`	initial start condition
`BEGIN`	condition switch start condition

lex: Working Example

myscanner.h

```
#define TYPE 1
#define NAME 2
#define PORT 3
#define COLON 4
#define IDENTIFIER 5
#define INTEGER 6
```

myscanner.l

```
%{
#include "myscanner.h"
%}

%%
:                   return COLON;
"db_type"           return TYPE;
"db_name"           return NAME;
"db_port"           return PORT;

[a-zA-Z][_a-zA-Z0-9]*    return IDENTIFIER;
[1-9][0-9]*              return INTEGER;
[ \t\n]             ;
.                   printf("unexpected
character\n");

%%
```

lex: Working Example

myscanner.c

```
#include <stdio.h>
#include "myscanner.h"

extern int yylex();
extern int yylineno;
extern char* yytext;

char *names[] = {NULL, "db_type", "db_name", "db_port"};

int main()
{
  int ntoken, vtoken;
  ntoken = yylex();
  //(continued in the next slide)
```

lex: Working Example

myscanner.c

```
  while(ntoken){
    printf("token type = %d\n", ntoken);
    if(yylex() != COLON){
        printf(" Syntax error in line %d, Expected a ':' but
found %s\n", yylineno, yytext);
        return 1;
    }
    vtoken = yylex();
    switch(ntoken){
        case TYPE:
        case NAME:
            if(vtoken != IDENTIFIER){
                printf("Syntax error in line %d, Expected an
identifier but found %s\n", yylineno, yytext);
                return 1;
            }
            printf("%s is set to %s\n", names[ntoken], yytext);
            break;
    }
  }
}
```

lex: Working Example

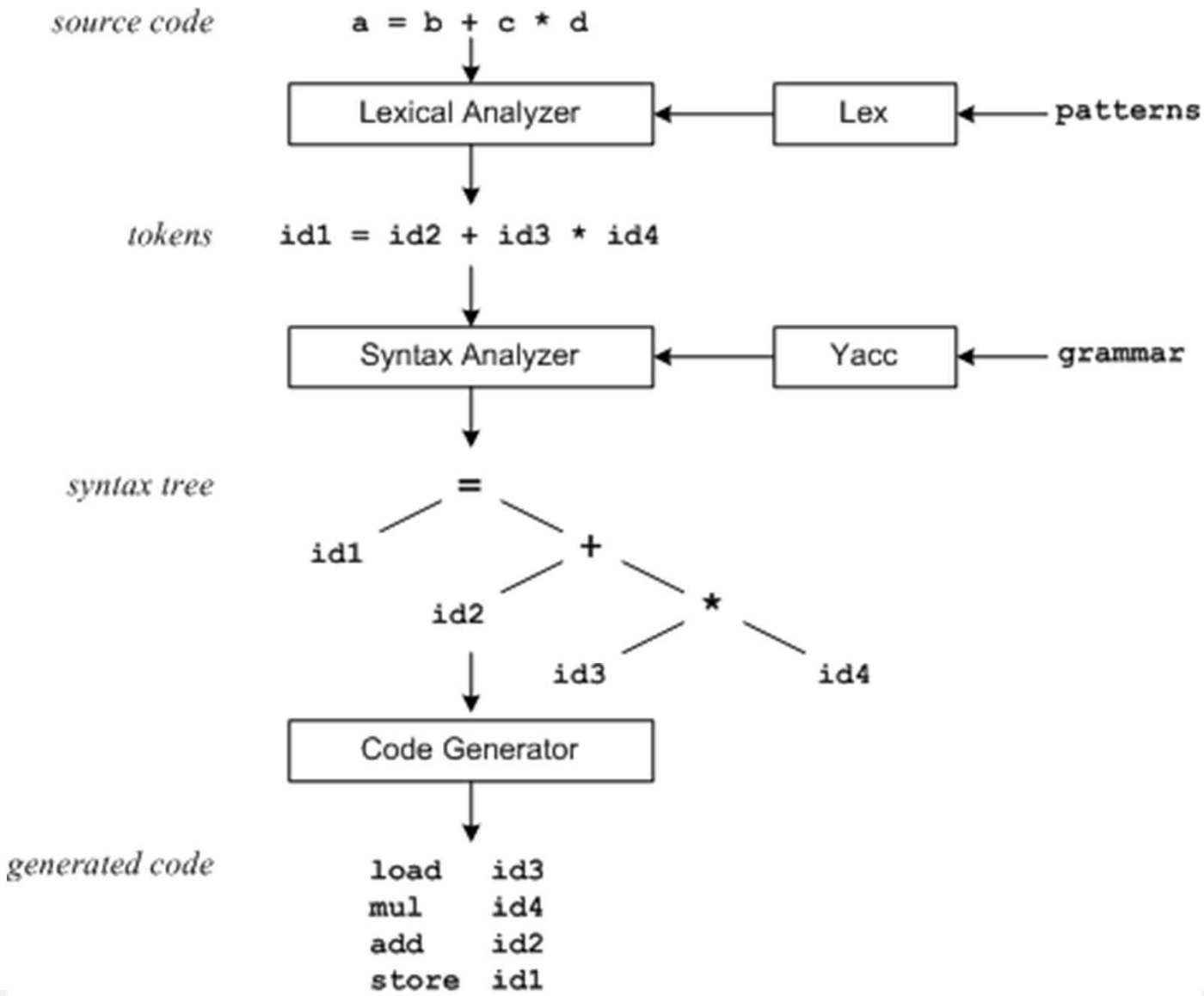
- $ gcc myscanner.c lex.yy.c -o myscanner
- $ ./myscanner < input.txt

lex / yacc

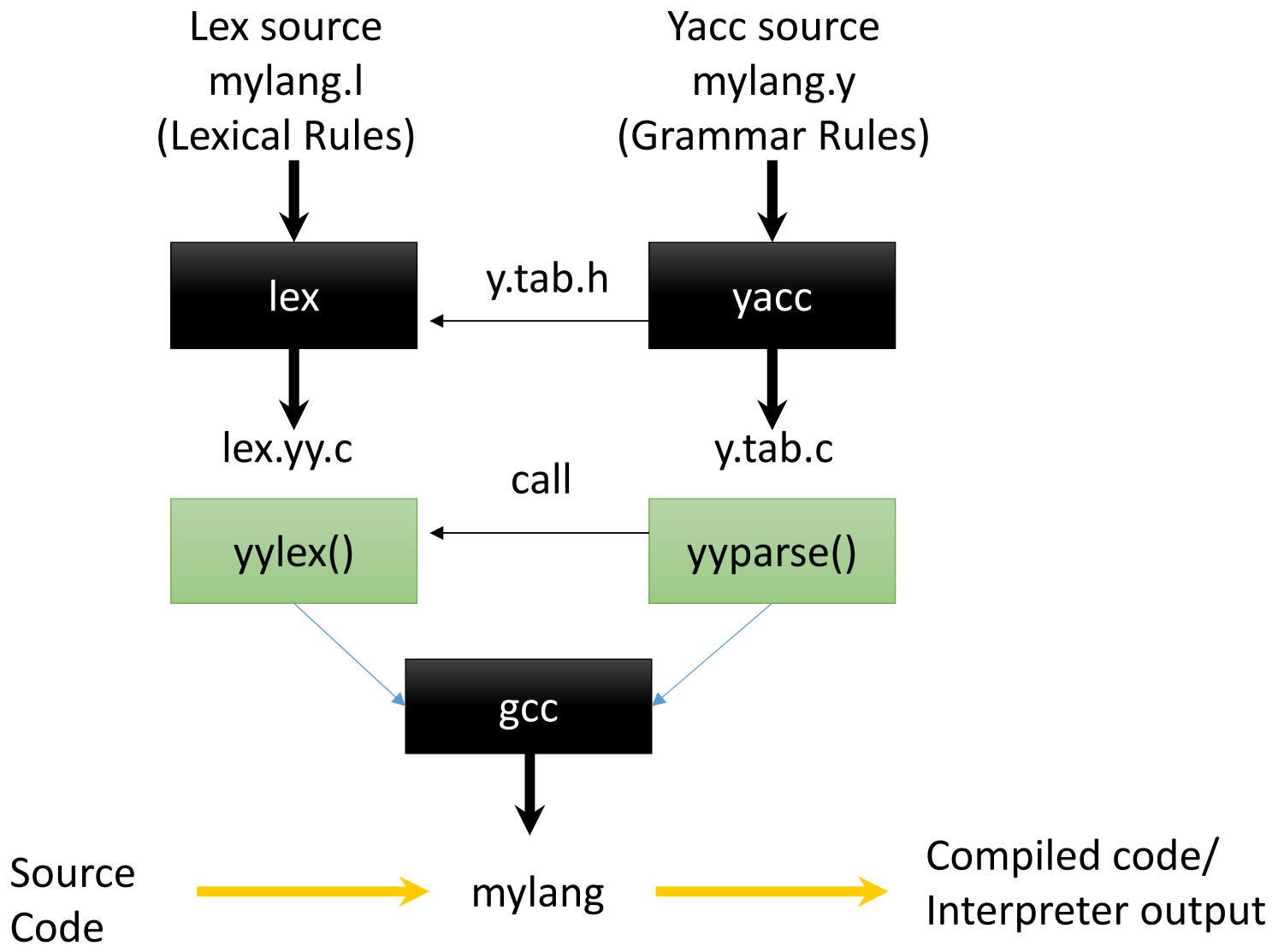
- lex
  - lex generates C code for a lexical analyzer, or scanner
  - lex uses patterns that match strings in the input and converts the strings to tokens

- Yacc (**y**et **a**nother **c**ompiler **c**ompiler)
  - Yacc generates C code for syntax analyzer, or parser.
  - Yacc uses grammar rules that allow it to analyze tokens from lex and create a syntax tree.
  - bison: GNU parser parser, upward compatibility with yacc

lex / yacc


source code
a = b + c * d
Lexical Analyzer
Lex
patterns
tokens
id1 = id2 + id3 * id4
Syntax Analyzer
Yacc
grammar
syntax tree
=
id1
+
id2
*
id3
id4
Code Generator
generated code
load id3
mul id4
add id2
store id1

Lex with Yacc



yacc input

- Structure of yacc input

```
FIRST PART
%%
production     action
...
%%
USER SUBROUTINES
```

- First part
  - C declarations enclosed in %{ %}
  - yacc definitions
    - %start → The first non-terminal in the grammar
    - %token → terminals
    - %union → declare the collection of data types
    - %type → declare the type of semantic values for non-terminals
    - %right, %left → declare the associativity associativity of tokens
    - %nonassoc → declare a token is not associative

yacc input

- Structure of yacc input

```
FIRST PART
%%
production     action
...
%%
USER SUBROUTINES
```

- Prodcutions
  - Left-hand side of a production is followed by a colon, and a right-hand side
  - Multiple right-hand sides may follow separated by a '|'
  - Actions associated with a rule are entered in braces

yacc example productions

```
Statements: statement
                {printf("statement");}
          | statement statements
                {printf("Statements");}

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

term: '(' expr ')'  {$$ = $2; }
    | ID
    | NUM
    ;
```

yacc productions

- `$1, $2, …, $n` refers to the values associated with symbols
- `$$` refers to the value of the left
- Every symbol has a value associated with it
  - Incuding token and non-terminals
  - Default action:
- `$$ = $1`

yacc input

- Structure of yacc input

```
FIRST PART
%%
production     action
...
%%
USER SUBROUTINES
```

- Third part
  - Contains valid C code that supports the language processing
    - Symbol table implementation
    - Functions that might be called by actions associated with the productions in the second part

Yacc with Lex - an example

```
<calc.l>
/* Definitions */
%{
#include <stdlib.h>
#include "calc.tab.h"
%}
NUMBER [0-9]+
%%
/* Rules */
{NUMBER} { yylval = atoi(yytext);
           return NUMBER; }
"+"      { return PLUS; }
"*"      { return MULT; }
"\n"     { return EOL; }
.    { yyerror("unexpected char"); }
%%
/* User Code */
```

```
<calc.y>
/* Definitions */
%{
#include <stdio.h>
%}
%token NUMBER PLUS MULT EOL
%%
/* Rules */
goal: eval goal  {}
     | eval      {}
     ;
eval: expr EOL { printf("= %d\n", $1); }
     ;
expr: NUMBER { $$ = $1; }
     | expr PLUS expr { $$ = $1 + $3; }
     | expr MULT expr { $$ = $1 * $3; }
     ;
%%
/* User Code */
int yyerror(char *s)
{   return printf("%s₩n", s);   }
```

Associativity & Precedence

- Specify associativity & precedence

```
%token NUMBER PLUS MULT EOL ASSN
```

```
%token NUMBER EOL
%left PLUS
%left MULT
%right ASSN
```

Precedence level

Lowest

Highest

- Change the grammar rules

```
expr0:   NUMBER { $$ = $1; }
         ;
expr1:   expr0 { $$ = $1; }
         | expr1 MULT expr0 { $$ = $1 * $3; }
         ;
expr2:   expr1 { $$ = $1; }
         | expr2 PLUS expr1 { $$ = $1 + $3; }
         ;
```

- If-else conflict can be resolved in yacc by specifying precedence

Precedence for If-Then and If-Then-Else

```
<clang.y>
/* Definitions */
%token NUMBER PLUS MULT EOL

%left PLUS
%left MULT

%nonassoc IF_THEN
%nonassoc ELSE

%%
/* Rules */
stmt : …
  | IF '(' expr ')' stmt %prec IF_THEN  { … }
  | IF '(' expr ')' stmt ELSE stmt  { … }
   …
%%
/* User Code */
 …
```

Tips

- For Lex (flex)
  - Single character can be used as token
  - char c = yytext[0];
  - String should be duplicated to safely pass outside the scanner
  - yylval.str = strndup(yytext, yyleng);
- For Yacc (bison)
  - Token and nonterminal symbol have type

```
%union {
    Exp* exp;
    char* str;
}
%token <str> ID
%type <exp> expr
```

Bison on the run

- $ ls
- calc.l calc.y main.c
- $ flex calc.l
- $ ls
- calc.l calc.y lex.yy.c main.c
- $ bison –d calc.y
- $ ls
- calc.l calc.y lex.yy.c calc.tab.c calc.tab.h main.c
- $ gcc -o calc.exe calc.tab.c lex.yy.c main.c –lfl
- $ ls
- calc.l calc.y lex.yy.c calc.tab.c calc.tab.h main.c
- calc.exe
- $ ./calc.exe

```
<main.c>
main()
{   yyparse();   }
```