Lex & Yacc
(GNU distribution - flex & bison)

Hwansoo Han

Prerequisite

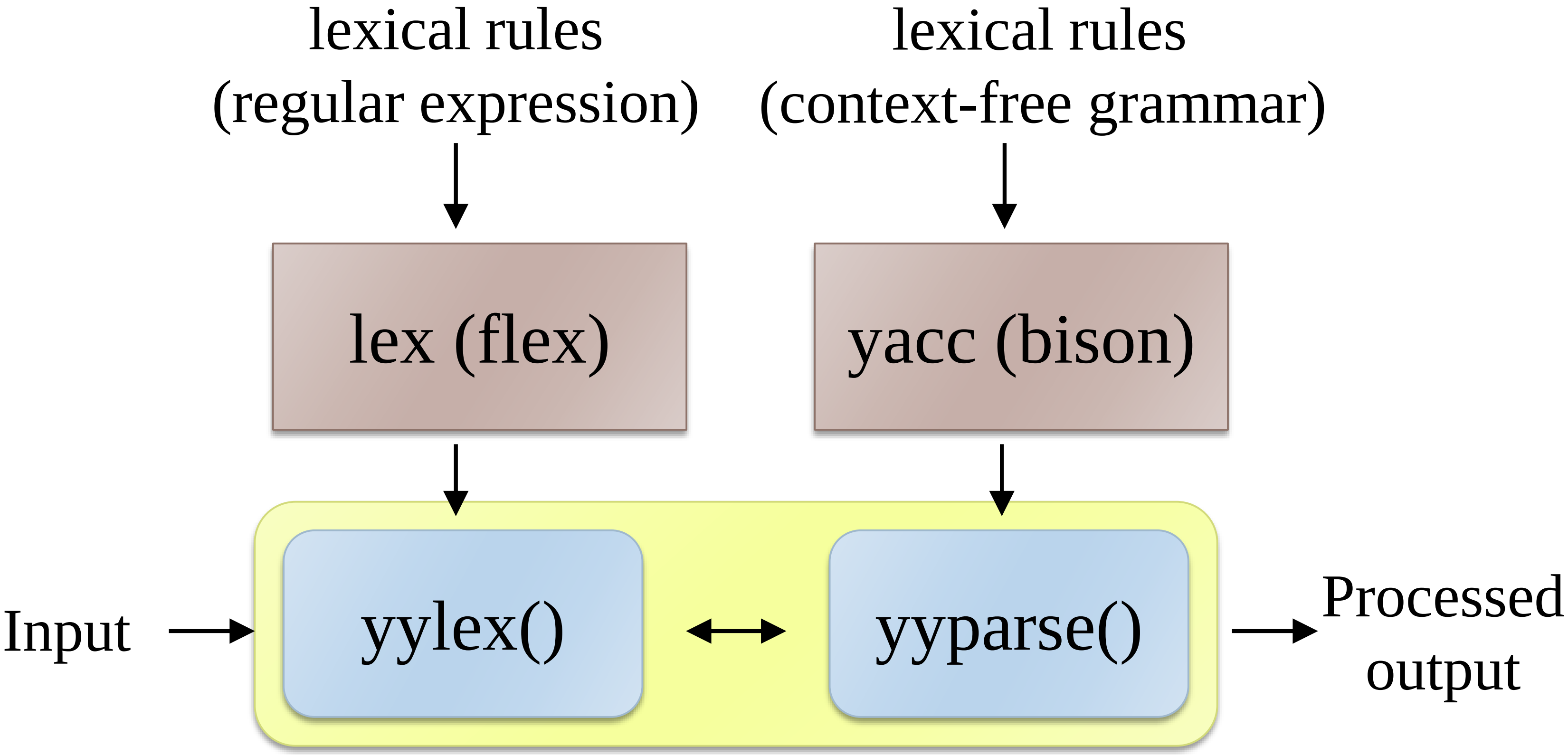
- Ubuntu
  - Version 14.04 or over
  - Virtual machine for Windows user or native OS
- flex
- bison
- gcc
  - Version 4.7 or over

- Install in Ubuntu
  - sudo apt-get install flex bison gcc

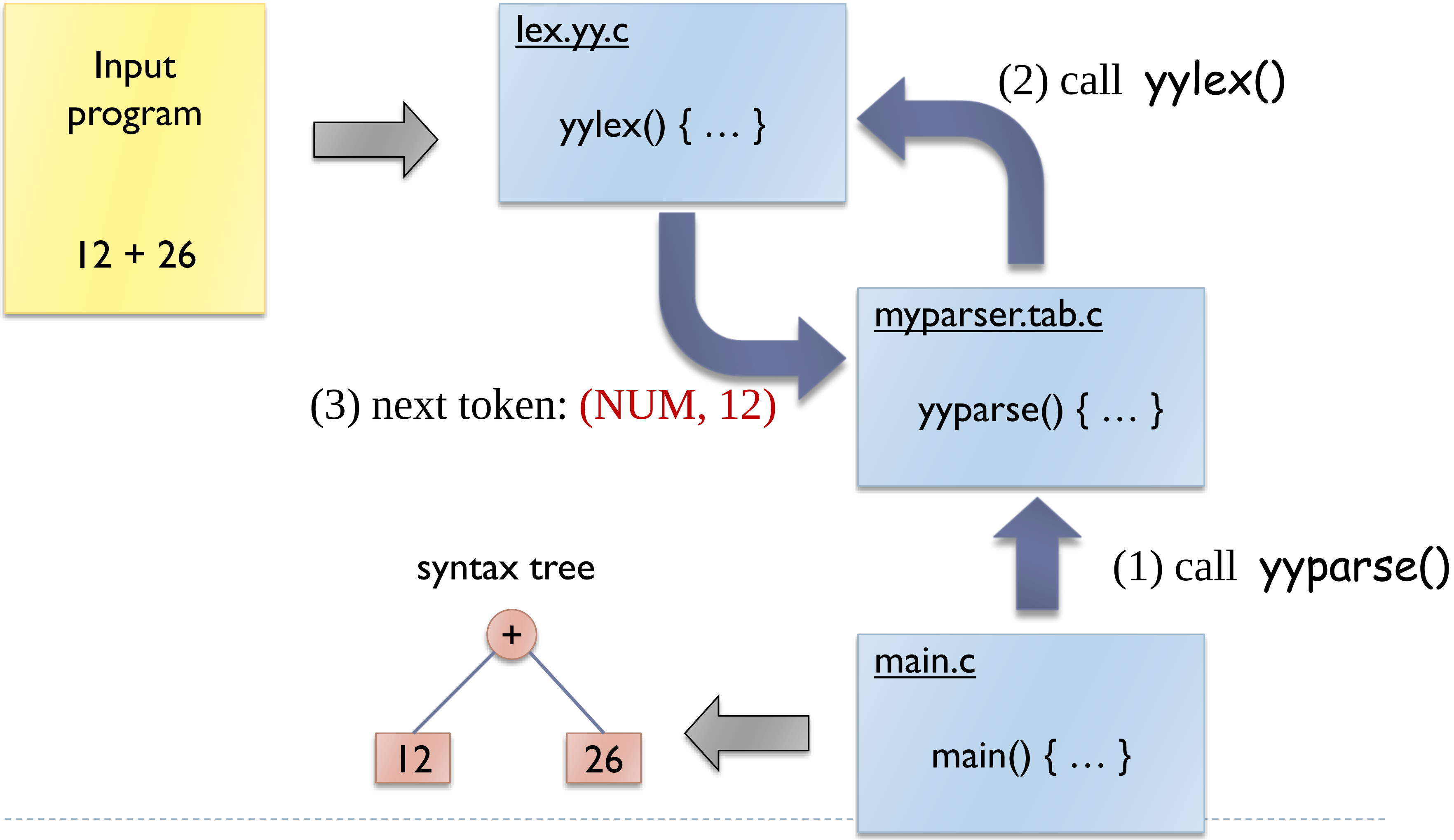
Flex Bison source code

- Flex
  - Input: sample.l
  - Output: lex.yy.c
  - Execution command
    - $> flex sample.l
- Bison
  - Input: sample.y
  - Output: sample.tab.c sample.tab.h
  - Execution command
    - $> bison -d sample.y

Lex & Yacc (flex & bison)



Working Version of Automated Software



Input program

12 + 26

lex.yy.c

yylex() { … }

(2) call yylex()

myparser.tab.c

yyparse() { … }

(3) next token: (NUM, 12)

(1) call yyparse()

main.c

main() { … }

syntax tree

Lex – A Lexical Analyzer Generator

- Source
  - A table of regular expressions and corresponding program fragments
  - Optional user codes

- Output: lex.yy.c (yylex())
  - The table is translated to a program which
    - Reads an input stream
    - Copy the input to an output stream
    - Partition the input strings which match the given expressions
    - As each such string is recognized the corresponding program fragment is executed
  - User codes are simply copied to the output

Lex – Format of Input File

```
%{
#include <stdio.h>
#include <stdlib.h>
#include “bison.tab.h”
void mycode(char *);
%}
```

headers

definitions

name   *definition*

```
ALPHA      [A-Za-z]
DIGIT      [0-9]
```

Units

```
%%
```

pattern   *action*

```
({ALPHA})({ALPHA}|{DIGIT})*    {
                                     printf(“ID = %s\n“, yytext);
                                     return LETTER;
                               }

({DIGIT})+                     mycode(yytext);
```

rules

```
%%

void mycode(char* text) {
    printf(“NUMBER = %d\n“, atoi(yytext));
    return NUMBER;
}
```

user codes

Flex on the run

```
$ cat main.c
main()
{
        yylex();
}
```

```
$ ls
   dn.l      main.c        sample.c

$ flex dn.l
$ ls
   dn.l      lex.yy.c      main.c        sample.c

$ gcc lex.yy.c main.c -lfl
$ ls
   a.out     dn.l   lex.yy.c     main.c        sample.c

$ ./a.out
…
Ctrl-D

$ ./a.out < sample.c
```

Regular Expressions

- . matches any single character
- * matches zero or more copies of preceding expression
- + matches one or more copies of preceding expression
- ? matches zero or one copy of preceding expression
    - -?[0-9]+ : signed numbers including optional minus sign
- [] matches any character within the brackets
    - [Abc1], [A-Z], [A-Za-z], [^123A-Z]

exclude [123A-Z]

- ^ matches the beginning of line
- $ matches the end of line
- \ escape metacharacter e.g. \* matches with *
- | matches either the preceding expression or the following
    - abc|ABC
- () groups a series of regular expression
    - (123)(123)*

Yacc – Yet Another Compiler-Compiler

- Similar to Lex

- Input: Context Free Grammar (CFG)

```
    Definitions
%%
    Grammar Rules
%%
    User Codes
```

- Output: <filename>.tab.h, <filename>.tab.c (yyparse())

Yacc – Analysis of AST

```
%{
#include <stdio.h>
#include “AST.h”
%}
```

headers

definitions

struct_type token

```
%type <ptr_Letter> LETTER
%type <ptr_Digit>  DIGIT
```

Declarations

```
%%
```

pattern *action*

```
LETTER : ..            { ... }

DIGIT  : ..            {
                           yyerror (“this is error”);
                       }
%%
```

Grammar Rules

```
void yyerror (char* text) {
    return fprintf(stderr, “%s\n“, text);
}
```

user codes

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 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

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

```
$ 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.out  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.out
$ ./calc.out
```

Symbol Table

```
Symbol Table

Location : main
  Count    Type  Name  Array       Role
      1     int  argc      -  parameter
      2   float     f      4   variable

Location : main - For(1) - If(1)
  Count    Type  Name  Array       Role
      1     int     a      -   variable
```