Guide to Yacc and Lex

Introduction

- Yacc: a parser generator
  - Describing the input to a computer program.
  - Take action when a rule matched.

- Lex: a lexical analyser generator
  - Recognise regular expression
  - Take action when one word matched

Example

- Configuration file
  - e.g. config.ini

```
ID = 42
Name = EvanJiang
.
.
.
```

Parsing. v1

- scan file rigidly

```
if (fscanf(parfile, "ID = %s\n", seed) != 1) {
   fprintf(stderr, "Error reading 'Seed:'\n");
   exit(0);
}
```

better choice ?

Parsing. better choice

- parsing tools accept the format like :

```
LIST '=' VALUE {
    $$ = $3;  /*$$ is result, $3 is the value of "VALUE"*/
}
```

YACC and LEX do this work well!

Yacc Overview

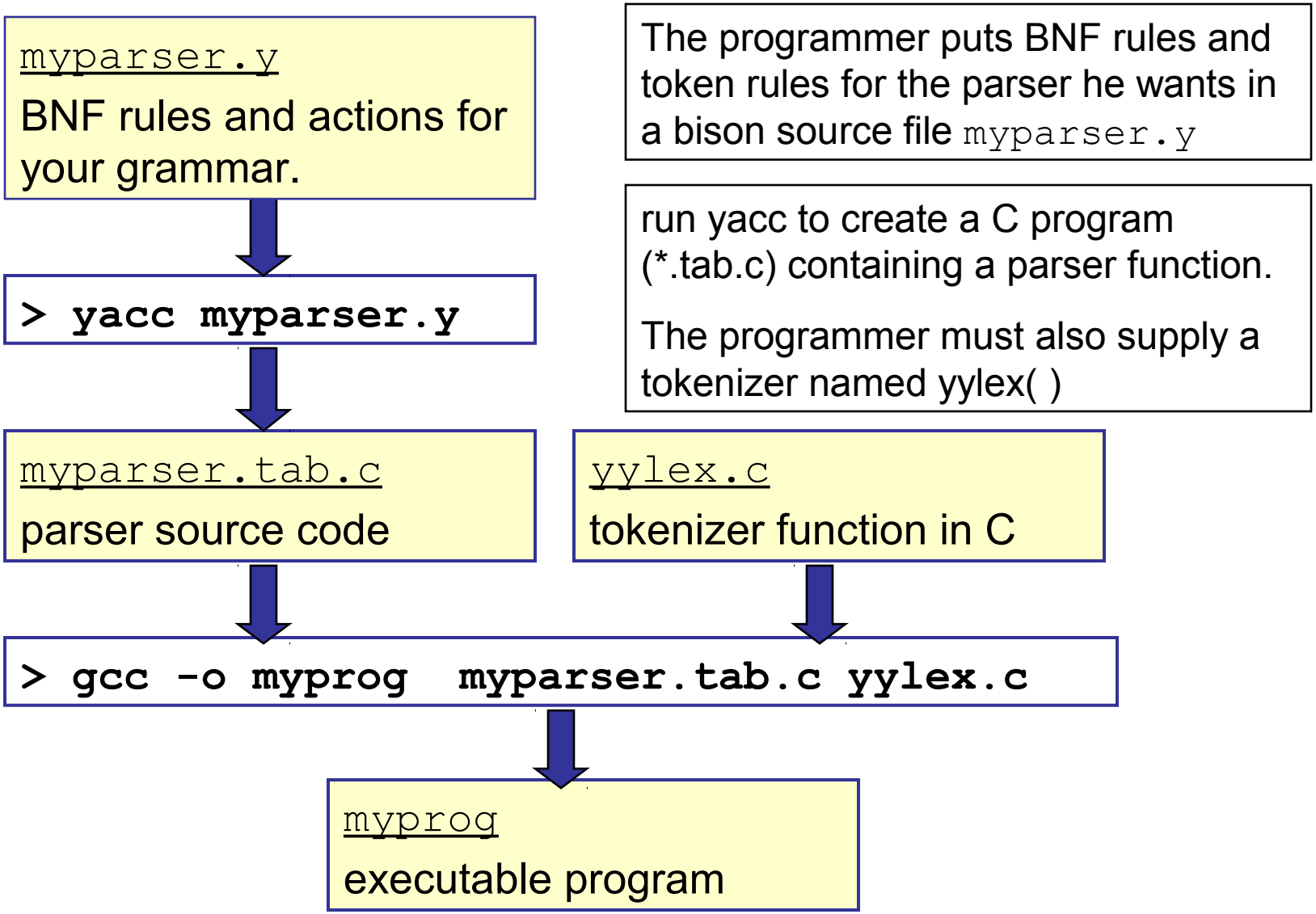
Purpose: automatically write a parser program for a grammar written in BNF.

Usage: you write a yacc source file containing rules that look like BNF.

Yacc creates a C program that parses according to the rules

```
term    : term '*' factor { $$ = $1 * $3; }
        | term '/' factor { $$ = $1 / $3; }
        | factor          { $$ = $1; }
        ;
factor  : ID              { $$ = valueof($1); }
        | NUMBER          { $$ = $1; }
        ;
```

Yacc Overview(2)



Yacc Overview(3)

In operation:

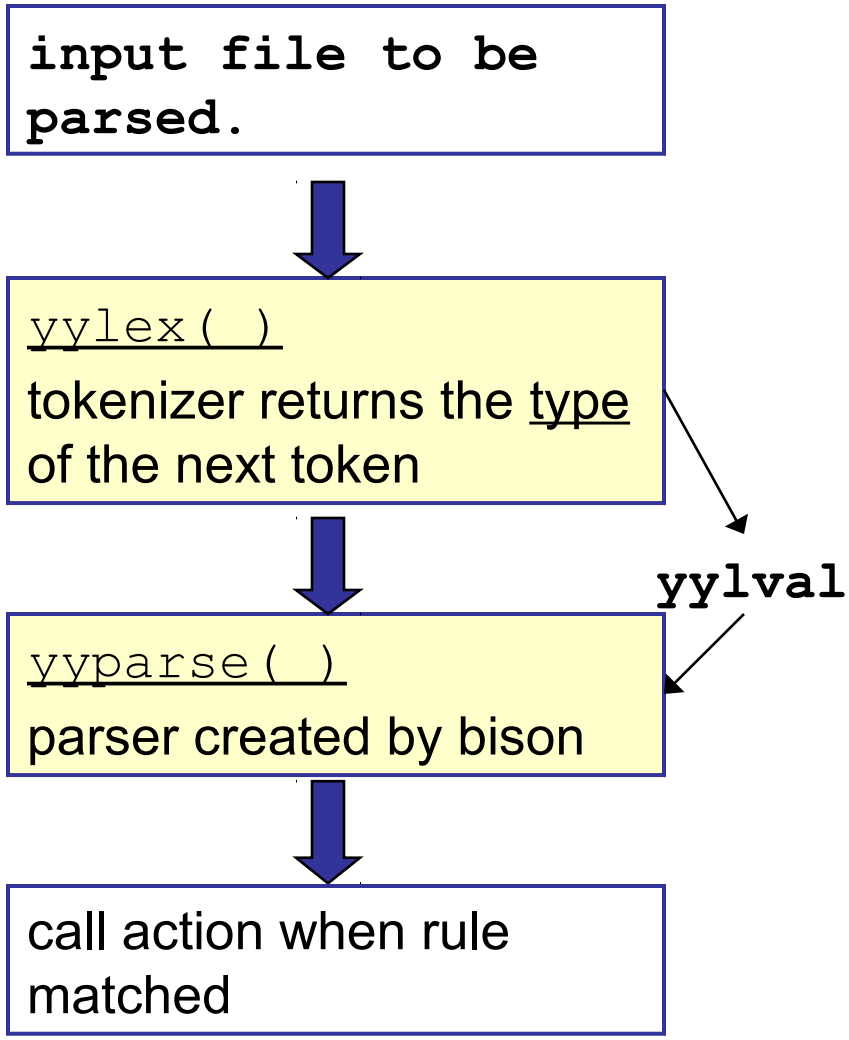
your main program calls `yyparse( )`.

`yyparse( )` calls `yylex()` when it wants a token.

`yylex` returns the **type** of the token.

`yylex` puts the **value** of the token in a global variable named `yylval`

yyparse() call action when one rule matched



Yacc source file

The file has 3 sections, separated by "%%" lines.

```
/* declarations go here */

%%
/* grammar rules go here */

%%
/* additional C code goes here */
```

Note: format for "yacc" is the same as for bison.

Yacc source file example

Structure of Bison or Yacc input:

```
%{
/* C declarations and #DEFINE statements go here */
  #include <stdio.h>
  #define YYSTYPE double
%}
/* Bison/Yacc declarations go here */
%token NUMBER      /* define token type NUMBER */


%%
/* grammar rules go here */
%%
/* additional C code goes here */
```

Provide by yylex(), which abstracts detail to a token

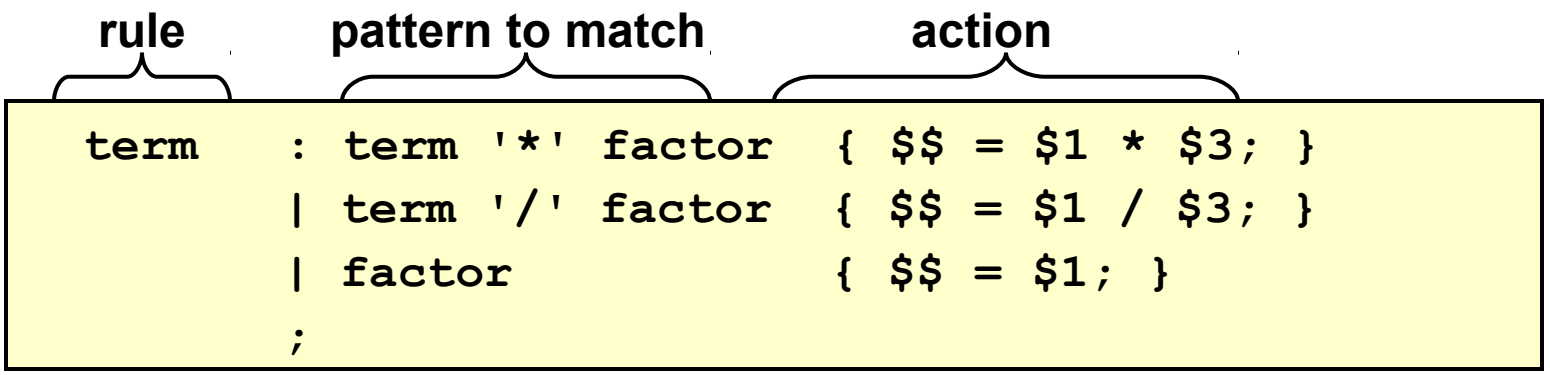
Yacc source file example(2)

```
%%    /* Bison grammar rules */
input    : /* empty production to allow an empty input */
         | input line
         ;
line     : term '\n'    { printf("Result is %f\n", $1); }

         ;
term     : term '*' factor { $$ = $1 * $3; }
         | term '/' factor { $$ = $1 / $3; }
         | factor          { $$ = $1; }
         ;
factor   : NUMBER          { $$ = $1; }
         ;
```

Yacc source file example(3)

- $1, $2, ... represent the actual values of tokens or non-terminals (rules) that match the production.
- $$ is the result.



Example:
if the input matches **`term / factor`** then set the result (**`$$`**) equal to the value of **`term`** divided **`factor`** (**`$1`** / **`$3`**).

Further studying

- Yacc with ambiguous grammar
  Precedence / Association
- Conflicts
  - shift/reduce conflict
  - reduce/reduce Conflicts
- Debug

Introduction to Lex

```
/* Bison/Yacc declarations go here */
%token NUMBER        /* define token type NUMBER */

factor   : NUMBER              { $$ = $1; }
         ;
```

- NUMBER,is given by Lex.
- Yacc calls yylex() to get the token and vale.

Introduction to Lex. cont.

- Lex is a program that ***automatically*** creates a scanner in C, using rules for tokens as regular expressions.
- Format of the input file is like Yacc.

```
%{
        /* C definitions for scanner */
%}
flex definitions
%%
rules
%%
user code (extra C code)
```

Regular Expression example

Regular Expression	Strings in L(R)
digit = [0-9]	"0" "1" "2" "3" …
posint = *digit*+	"8" "412" …
int = -? *posint*	"-42" "1024" …
[a-zA-Z_][a-zA-Z0-9_]*	C identifiers

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
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

Lex example

- Read input and describes each token read.

```
/* flex definitions */
DIGIT    [0-9]
%%

[ \t\n]+   {}
-?{DIGIT}+  { printf("Number: %s\n", yytext);
              yylval=atoi(yytext);   return NUMBER; }
\n          printf("End of line\n"); return 0;
%%

/* all code is copied to the generated .c file*/
```

Example explanation

```
/* flex definitions */
DIGIT    [0-9]
%%

[ \t\n]+    {}
-?{DIGIT}+  { printf("Number: %s\n", yytext);
              yylval=atoi(yytext);   return NUMBER; }
\n          printf("End of line\n"); return 0;
%%

/* all code is copied to the generated .c file*/
```

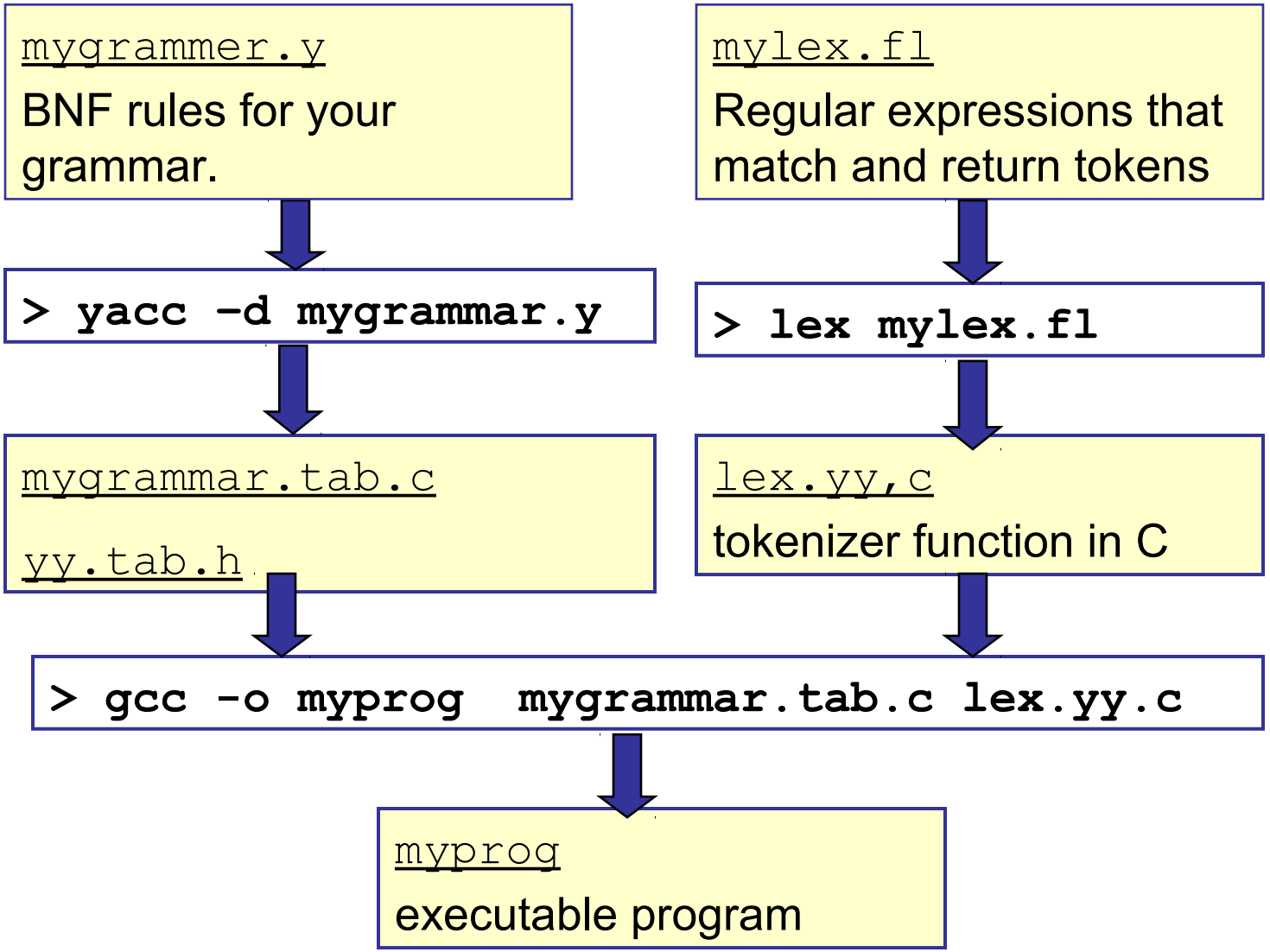
Yacc get the value

Yacc get the token

Further studying

- Regular expression
- Debug

Review


mygrammer.y
BNF rules for your grammar.
mylex.fl
Regular expressions that match and return tokens
> yacc -d mygrammar.y
> lex mylex.fl
mygrammar.tab.c
yy.tab.h
lex.yy.c
tokenizer function in C
> gcc -o myprog mygrammar.tab.c lex.yy.c
myprog
executable program

暨南大學
JINAN UNIVERSITY

Thanks