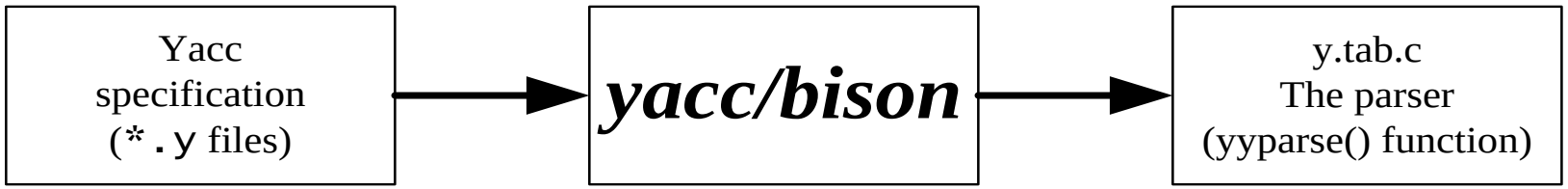
Yacc

What is Yacc

- Yacc: Yet Another Compiler-Compiler

- A LALR(1) parser generator
  - Read rules from a specification file (`*.y` file)
  - Using *.y to Generate the lexical analyzer in C language
    - `*.tab.c` and `*.tab.h` file
    - Contain a `yyparse()` function – the parser
    - **bison** is a later implementation of **yacc.**



Simple compiler

- A compiler can scan the character stream into tokens and then analyze the syntax of the token stream, but Yacc can only analyze the syntax .

How to complete a compiler ?

- Using Lex to scan characters into tokens and Yacc to analyze the syntax of the token stream.

Cooperation of Yacc and Lex

- Using .y file to generate *.tab.c and *.tab.h file

```
bison –d -o “file_name”.tab.c “file_name”.y
```

```
gcc -c -g -I.. “file_name”.tab.c
```

- Using .l file to generate *.yy.c file

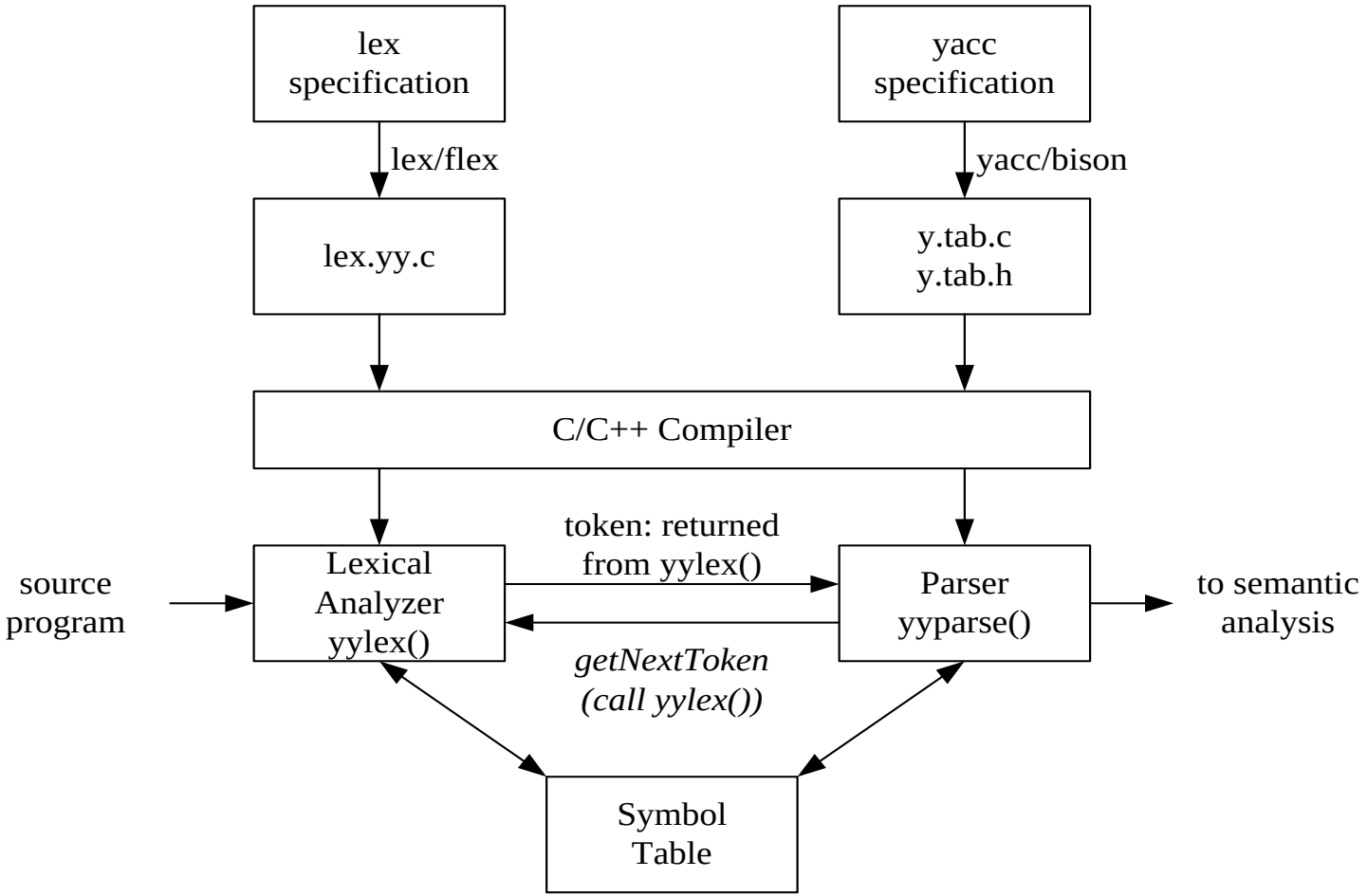
```
flex –o “file_name”.yy.c “file_name”.l
```

```
gcc -c -g -I.. “file_name”.yy.c
```

- Link .c generated by yacc and lex

```
gcc -o “file_name” “file_name”.tab.o “file_name”.yy.o -ll
```

Cooperation of yyparse() and yylex()



Example code
加法分析器
ex1.l

```
%{
#include "y.tab.h"
%}
%%
\n              { return(0); /* EOF */ }
[ \t]+          { /* do nothing */ }
[0-9]+          { yylval.ival = atoi(yytext);return(INUMBER); }
"+"|"("|")"     { return(yytext[0]); }
.               { return(yytext[0]); }
%%
```

Lex part of Cooperation

- *.l file need to include the *.h file , which is actually generated by yacc/bison
  - #include "y.tab.h“
- Lex return token’s values to Yacc
  - yylval.ival = atoi(yytext);
  - return(INUMBER);
  - return(yytext[0]);
- yylval
  - This is a variable defined in *.y file
  - When token is returned to yacc , yacc can use it to do something

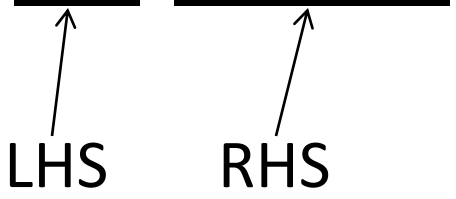
Example code
加法分析器
ex1.y

```
%{
#include <stdio.h>
#include <string.h>
void yyerror(const char *message);
%}
%union {
int ival;
}
%token <ival> INUMBER
%type <ival> expr
%left '+'
%%
line    : expr                { printf("%d\n", $1); }
        ;
expr    : expr '+' expr       { $$ = $1 + $3; }
        | INUMBER
        ;
%%
void yyerror (const char *message)
{
        fprintf (stderr, "%s\n",message);
}

int main(int argc, char *argv[]) {
        yyparse();
        return(0);
}
```

Bottom up Parser

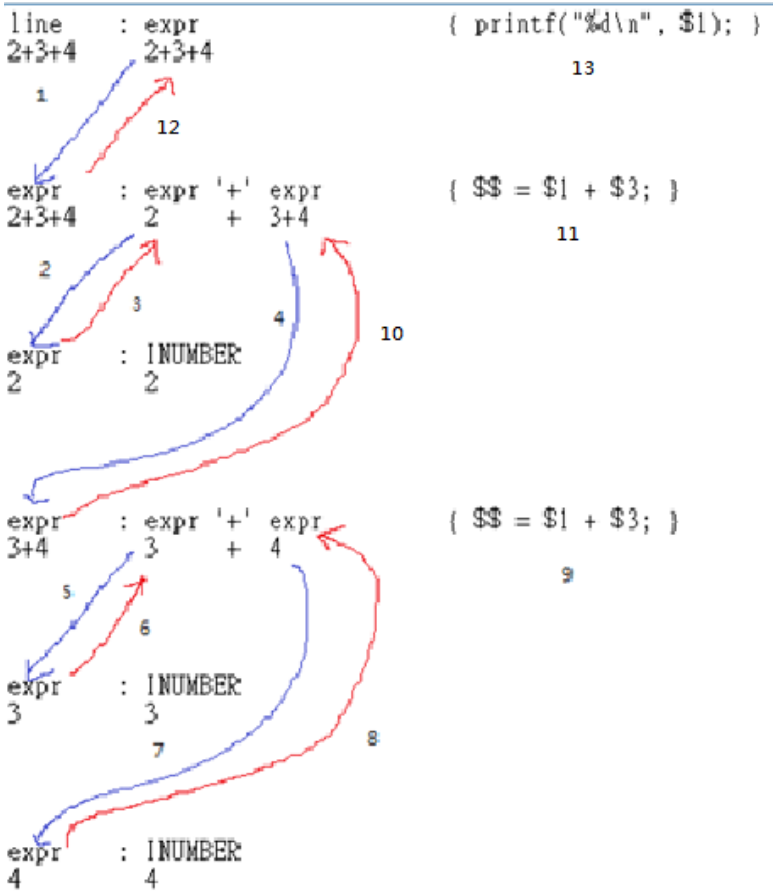
- When tokens match all of grammar's RHS(right hand side),the result of RHS will be reduce to grammar 's LHS(left hand side)
  - ex: expr: INUMBER

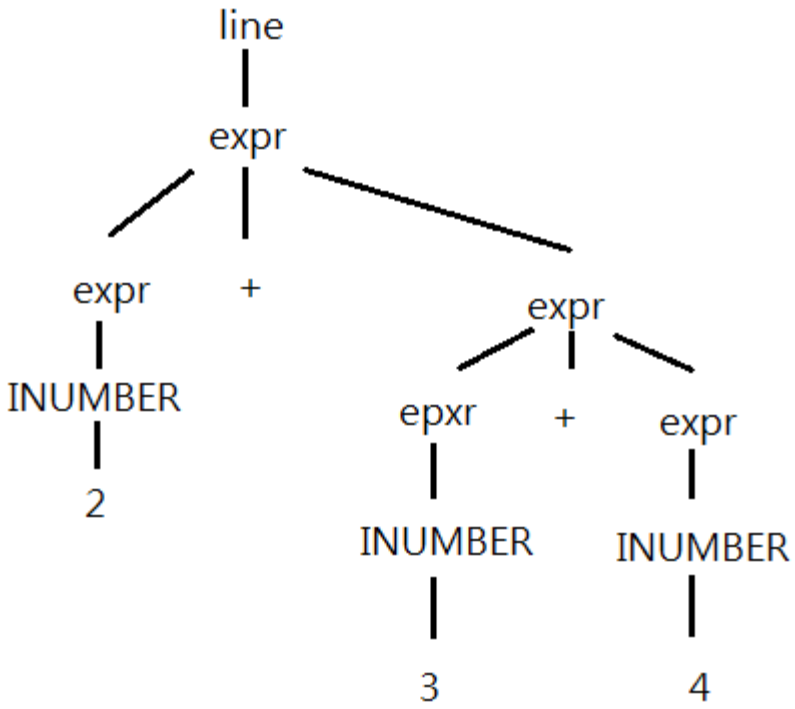


- Bottom up until reduce the start symbol
  - example code start symbol is line

Example code Process Analysis

- Ex:2+3+4





Compile and Run the Example

- How to compile?

```
#compile bison
bison -d -o y.tab.c ex1.y
gcc -c -g -I.. y.tab.c
#compile flex
flex -o lex.yy.c ex1.l
gcc -c -g -I.. lex.yy.c
#compile and link bison and flex
gcc -o ex1 y.tab.o lex.yy.o -ll
```

Run

$ ./ex1

$ 1+2+3

Output：

$ 6

Practice 1

- Use *.sh to execute compile command

The First Impression

- The structure of a yacc program – three sections
  - Definition section: any initial C/C++ codes or definitions
  - Rules section: grammar + action, separated by whitespaces. A production rule consists of grammar and action.
  - Subroutine section: any legal C or C++ codes

- Sections are separated by “%%” lines

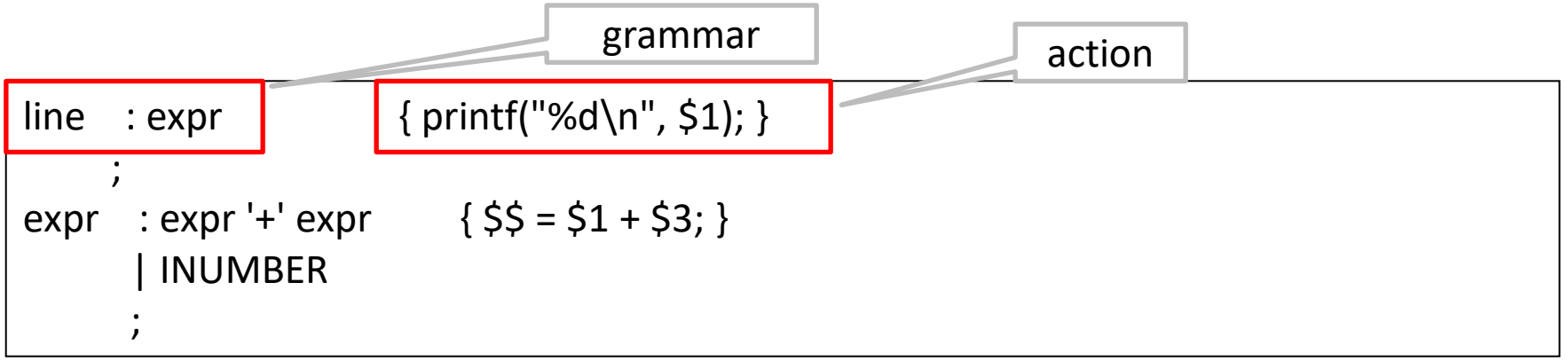
```
… Definition Section …
%%
… Rules Section …
%%
… Subroutine Section …
```

The Subroutine Section

- yyerror
  - deal with error . ex: sytax error
- yyparse
  - Use the productions of rule section to analyze the syntax
  - Communication with yylex

The Rules Section

- The context-free grammar used to generate the parser, the `yyparse()` function
- A production can contain grammar and action
- A production must be terminated with a semi-colon (;)
- The action is executed on matching a production
- '|' can merge productions having the same non-terminal at ':' left
- An example: expr : expr '+' expr |INUMBER



Embedded Rule Action

- We would like to do something when only parts of a production is matched
  - `prod1 : A B { printf("A B are seen"); } ;`
  - `prod1 : A { printf("A is between B"); } B ;`
    - This action will execute after A is analyzed done
- The above production is equivalent to
  - ```
    prod1   : A B epsilon ;
    epsilon : /*empty*/ { printf("A B are seen"); } ;
    ```
  - ```
    prod1   : A epsilon B;
    epsilon : /*empty*/ { printf("A is between B"); } ;
    ```

Symbol Attributes

- The attribute (value) associated with a symbol – yylval
  - In a Lex rule action, you can assign an attribute to a token

```
[0-9]+        { yylval = atoi(yytext); return(NUMBER); }
```

  - By default, yylval is of type int
  - By default, attributes associated with symbols are all type int
  - Attributes of symbols can be accessed by $$, $1, $2, $3 …

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

Non-Integer Symbol Attributes

- The %union directive
  - *%union{*
    datatype variable-name;
    …
    }
- yylval has all data type defined in %union,you can assign some data type to yylval
  - ex: yylval.ival = atoi(yytext); /*at *.l file*/
- Declare data type for terminals and non-terminals
  - terminals: %token <variable-name> token …
  - non-terminals: %type <variable-name> non-terminal …

Synthesized VS Inherited Attribute

- There are two types of symbol attribute for a non-terminal
  - Synthesized – depends only on itself and its children
  - Inherited – depends on its parent, itself, and it's siblings
- We have introduced the synthesized symbol attributes , which can be accessed by $$, $1, $2, …
  - ex in ex1.y: expr : expr '+' expr { $$ = $1 + $3; }
- How to use attributes from parents?
  - Access parent attributes using $0, $-1, …
  - However, as we cannot see parents in the production, we have to explicitly assign its data type using $<varname>0, $<varname>-1, …

Inherited Attribute – an Example

```
declaration  : class type namelist ;
class        : GLOBAL          { $$ = 1; }
             | LOCAL           { $$ = 2; }
             ;
type         : REAL            { $$ = 1; }
             | INTEGER         { $$ = 2; }
             ;
namelist     : NAME            { mksymbol($0, $-1, $1); }
             ;
```

```
mksymbol : set the name's type

ex: LOCAL REAL rvalue

    rvalue will be set in symbol table
```

class	type	namelist
2	1	rvalue

- Given the above grammar of variable declaration
  - If default type is integer
    - mksymbol($0, $-1, $1)
  - If variable name for class and type is .class and .type, respectively
    - mksymbol($<type>0, $<class>-1, $1)

Practice 2

- 此練習為熟悉action，並了解Inherited的使用時機
- 分析的語法為：字串 字串 數字
- .y file
  - line :test test integer 此production的action要輸出數字
  - integer :INUMBER 此production的action要輸出數字前的兩個字串
- .l file
  - [a-zA-Z]+ 此rule的action把yytext的值丟給yylval

Practice 2

*.l file

```
%{
#include "ex2_1.tab.h"
%}
%%
\n              { return(0); /* EOF */ }
[ \t]+          { /* do nothing */ }
[0-9]+          { yylval.ival= atoi(yytext); return(INUMBER); }
[a-zA-Z]+   {/*action of yytext value to yylval*/}
.               { return(yytext[0]); }
%%
```

Practice 2

*.y file

```
%{
#include <stdio.h>
#include <string.h>
void yyerror(const char *message);
%}
%union{
int ival;
char* word;
}
%token<ival> INUMBER
%token<word> WORD
%type<ival> integer
%type<word> test
%%
line :test test integer {/*action of output integer*/}
    ;
test : WORD
    ;
integer :INUMBER {/*action of output inherited attribute*/}
        ;
```

Practice 2

*.y file

```
%%
void yyerror(const char *message)
{
                fprintf(stderr, "%s\n",message);
}int main(int argc, char *argv[])
{
   yyparse();
   return(0);
}
```

Practice 2 hint

- Use strdup to change yytext type to yylval
  - ex: yylval = strdup(yytext);

Practice 2 input & output

Input:

uuu ttt 123

Output:

first: uuu second: ttt

123

The Definition Section

- Contains initial C or C++ codes
- C/C++ Codes must be enclosed with “%{“ and “%}” lines
- We will discuss other useful definitions later
- A skeleton for embedding initial codes

```
%{
initial C or C++ codes
%}
… Other Definitions …
%%
… Rules Section …
%%
… Subroutine Section …
```

Token Declarations

- %token: generic token declarations
- %left: left-associative binary operators with precedence

  ex: 9-3-2 = 4 not 8
- %right: right-associative binary operators with precedence

  ex: 2^2^3 = 256 not 64
- %nonassoc: non-associative tokens with precedence

Generic Token Declarations – %token

- Basic
  - *%token token1 [token2 token3 …]*
  - Ex: %token INTEGER DOUBLE STRING IF ELSE
- Token numbers are assigned by yacc automatically
  - Higher than any other possible character code (256+)
  - Not conflicting with any literal tokens
- Alternatively, you can manually assign the numbers
  - *%token token1 [number1] [token2 [number2] …]*
  - Ex: %token UP 50 DOWN 60 LEFT 17 RIGHT 25
- Note: do not use zero as the token number
  - yylex() returns zero on end of file

Operator Precedence and Associativity

- Declare operators in yacc
  - %left – left associative
  - %right – right associative
  - %nonassoc – non-associative
- An operator declaration line is able to contain multiple operators, which have the same precedence
  - Ex: %left ‘+’ ‘-’
- We declare operators in the order of their precedence, from lower to higher, for example
  - %left ‘+’ ‘-’
  - %left ‘*’ ‘/’
  - %right POW

Precedence Assignment – the %prec Directive

- A token may have a higher precedence than it used to be
- For example, the '+' and the '-' token
  - For a calculator, usually */ has higher precedence than +-
  - But, if +- are used as unary operator, i.e, -1, +2, it may have higher precedence than */
  - The below example gives '+' the highest precedence when '+' is a unary operator
    - The %prec directive gives the rightmost token (terminal) in the production an equivalent precedence to the specified token

```
%left '+'
%left '*'
%nonassoc UPLUS
%%
E       : E '+' E
        | E '*' E
        | '+' E  %prec UPLUS
        ;
```

Practice 3

- Simple calculator
  - calculate real number
  - ‘+’ ‘-’ ‘*’ ‘/’ operation
  - ‘(’ ‘)’ may appear on expression
  - Output to the first digit after the decimal point

Practice 3

- Input :

2.0*2.5

5.0*(3.5+2)

5.25*3

- Output:

5.0

27.5

15.8