CS340400 Compiler Design Homework 2

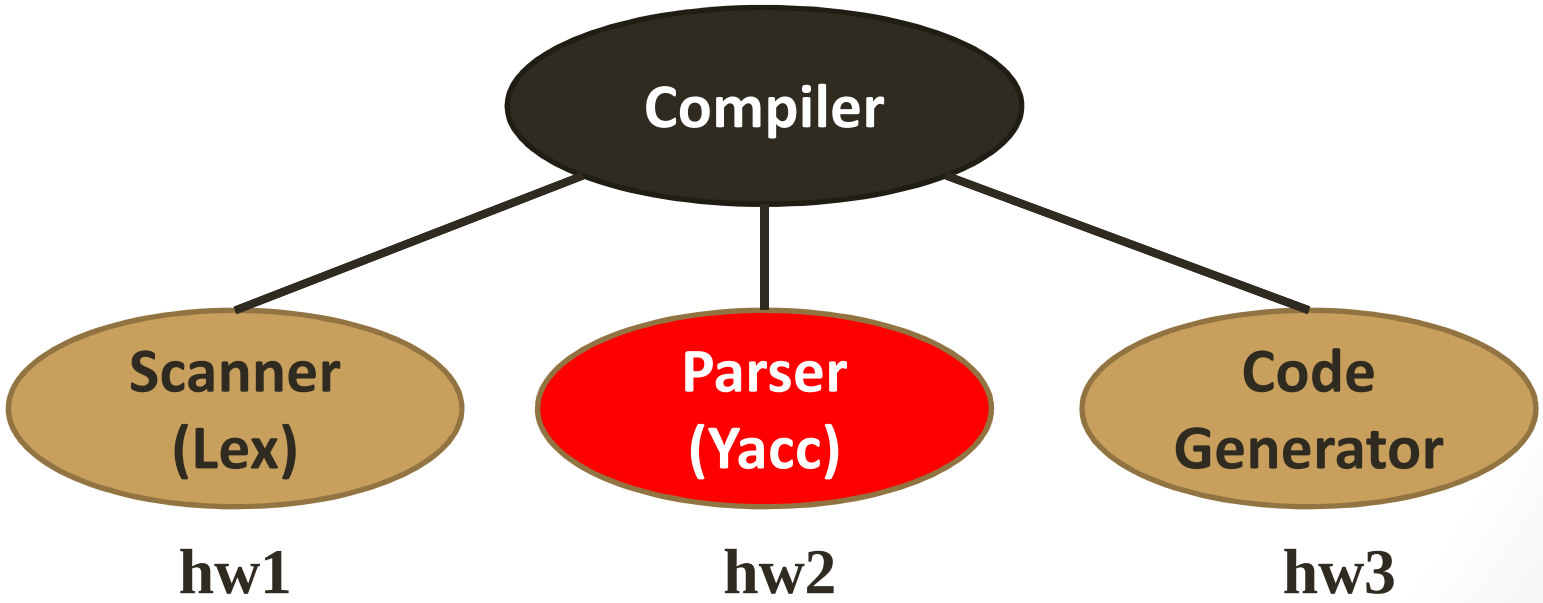
2015/04/23

Outline

- Yacc
- Homework2

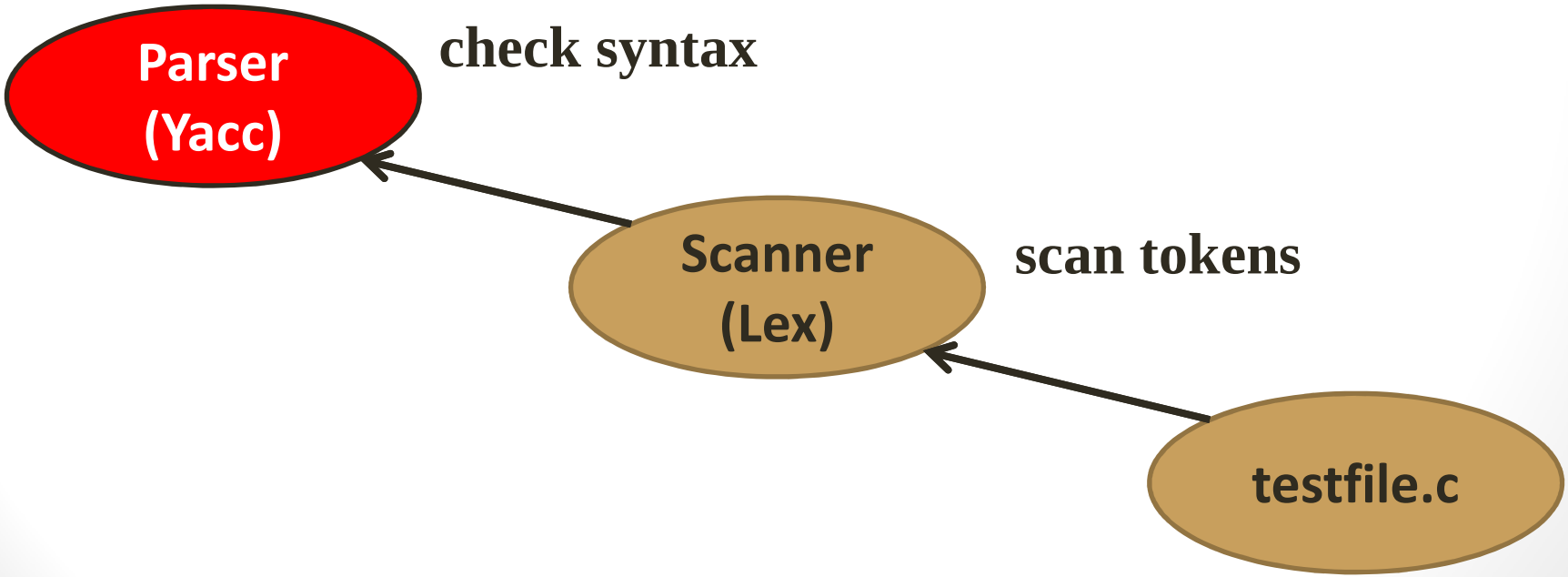
Why do we need Lex and Yacc?

- Writing a compiler is difficult, and it requires a lot of time and effort.
- Lex and Yacc help us to automate the process of constructing scanner and parser.



Introduction of Yacc

- Yacc (Yet Another Compiler Compiler)
- Yacc produces a parser with given grammar.
- Parser reads the tokens scanned by Scanner, and check whether they are correspond to the syntax.



Yacc file structure

- The file structure contains three parts.
  - (1)Definition section
  - (2)Grammar section
  - (3)User-defined function section
- The sections are separate with “%%”.
- Yacc is similar to Lex!

An Yacc file example

```
%{
#include <stdio.h>
%}

%token NAME NUMBER
%%

statement: NAME '=' expression
        | expression            { printf("= %d\n", $1); }
        ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
        |   expression '-' NUMBER { $$ = $1 - $3; }
        |   NUMBER                { $$ = $1; }
        ;
%%
int yyerror(char *s)
{
   fprintf(stderr, "%s\n", s);
   return 0;
}

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

%{

C declarations

%}

Yacc declarations

%%

Grammar rules

%%

Additional C code

Comments enclosed in /* ... */ may appear in any of the sections.

Definitions section

```
%{
#include <stdio.h>
#include <stdlib.h>
%}

%token NAME NUMBER
%start expr
```

Terminals:
The tokens return by Scanner(Lex)

The start non-terminal:
The default is the first non-terminal in grammar section.

Grammar section

Grammar：

expr → expr '+' term | term

term → term '*' factor | factor

factor → '(' expr ')' | ID | NUM

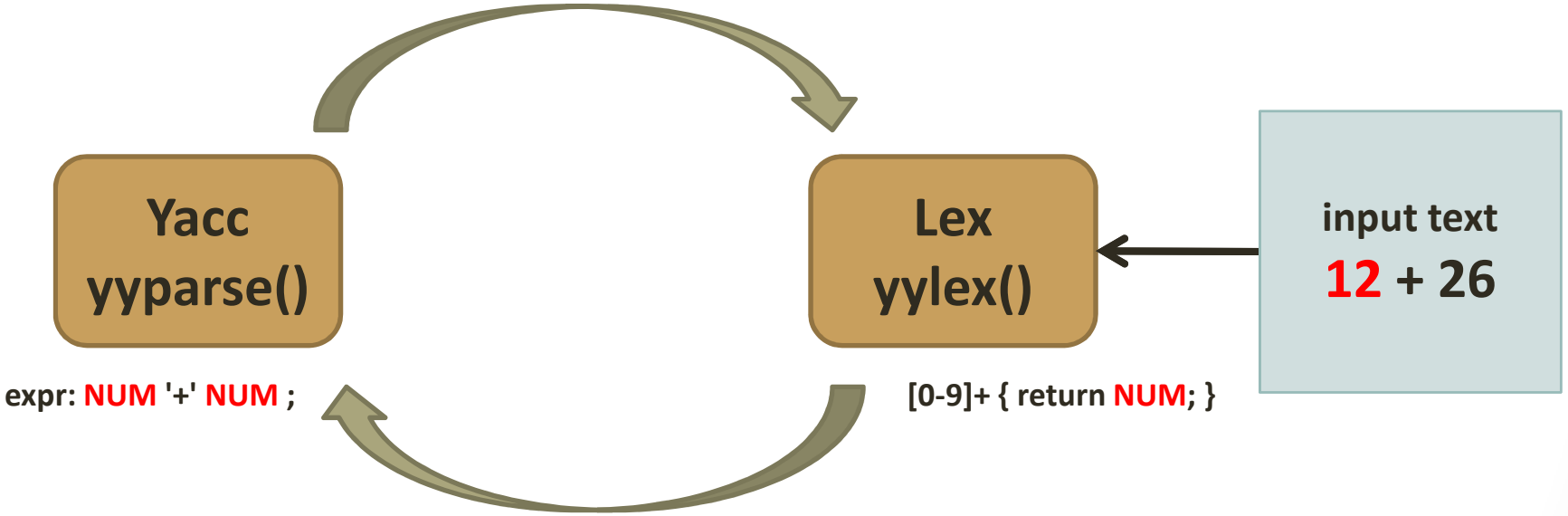
It should be written like this in Yacc. ↓

Yacc file：

```
expr   : expr '+' term
       | term
       ;
term   : term '*' factor
       | factor
       ;
factor : '(' expr ')'
       | ID
       | NUM
       ;
```

Communication between Lex and Yacc

yyparse() calls yylex() automatically



return token
the first token is NUM

Communication between Lex and Yacc

- In order to make communication between Lex and Yacc, we need an interface.
- The interface is y.tab.h file, which is produced by Yacc.

- How to create y.tab.h and use it?
  - $ byacc –d homework.y
    - The command will produce y.tab.h and y.tab.c.
  - Include y.tab.h in Lex program.

- Note: The name of the produced file may not be the same with different Yacc.

Communication between Lex and Yacc

test.l

```
%{
#include "y.tab.h"
%}
id          [a-zA-Z][a-zA-Z0-9]*
%%
int         { return INT; }
char        { return CHAR; }
float       { return FLOAT; }
{id}        { return ID;}
```

test.y

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token CHAR, FLOAT, ID, INT
%%
…
```

$ byacc -d test.y

will produce *y.tab.h*

The content of *y.tab.h*

```
# define CHAR 258
# define FLOAT 259
# define ID 260
# define INT 261
```

Use Lex to pass token value to Yacc

- Terminal symbol (token) may represent a value of a data type.
  - For example:
    - A numeric quantity like 42, or a pointer points to a string “Hello world!”.
- When we use Lex, we can put the value into **yylval**.
  - In more complex situation, **yylval** is an **union**.
- Lex:
  - code example:
  - [0-9]+ { yylval = atoi(yytext); return NUM; } // yylval is not an union
  - [0-9]+ { yylval.intVal = atoi(yytext); return NUM; } // yylval is an union
- Yacc:
  - Use $$, $1, $2 …… to get the value.

Use Lex to pass token value to Yacc

- Yacc allows tokens to have a value of different data type.
- The type of **yylval** is defined in Yacc with **%union**.

```
test.y
...
%union{
    int intVal;
    double douVal;
    char* strVal
}
%%
```

byacc –d test.y

```
y.tab.h
...
extern YYSTYPE yylval;
```

include y.tab.h
into Lex program

```
test.l
...
%{
#include “y.tab.h”
%}
...
%%
[0-9]+      { yylval.intVal = atoi(yytext); return NUM;}
[a-zA-Z]+   { yylval.strVal = strdup(yytext); return STRING; }
```

Use Lex to pass token value to Yacc

Use $$, $1, $2 …… to get the value from terminal/non-terminal symbol in Yacc.

```
expr   : expr  '+'  term        { $$ = $1 + $3; }
       | term                   { $$ = $1; }
       ;
term   : term  '*'  factor      { $$ = $1 * $3; }
       | factor                 { $$ = $1; }
       ;
factor : '('  expr  ')'         { $$ = $2; }
       | ID
       | NUM
       ;
```

$1 $2 $3

$$

note：

1. The default action is { $$ = $1; }.
2. You can do other things, like printf() in {}.

Yacc example

```
%{
#include "symtab.h"
#include <stdio.h>
%}

%union {
  double dval;
  struct symbol *symbol;
}

%token <symbol> ID
%token <dval> DOUBLE

%type <dval> statement_list
%type <dval> statement
%type <dval> term
%type <dval> factor

%%
```

```
statement_list -> statement |
                  statement_list statement

Statement -> ID '=' term |
             term

term -> term '*' factor |
        term '/' factor |
        factor

factor -> '(' term ')' |
          '-' factor |
          DOUBLE |
          ID
```

The data type of ID terminal is the same with *symbol* in the %union

The data type of all non-terminal is the same with *dval* in the %union

Yacc example

```
statement_list: statement '\n' {
              $$ = $1; printf("reduce to statement_list from statment:%f\n", $1);
             }
            | statement_list statement '\n' {
              $$ = $2;
              printf("reduce to statement_list from statement_list statment:%f\n", $2);
             }
            ;
statement: ID '=' term {
         $1->value = $3; $$ = 0;
         printf("reduce to statement from ID:%s = term:%f\n", $1->id, $3);}
        | term       {
         $$ = $1; printf("reduce to statement from term:%f\n", $1);}
        ;
term: term '*' factor {
        $$ = $1 * $3; printf("reduce to term from %f * %f\n", $1, $3);
      }
    | term '/' factor {
        if ($3 == 0.0) {
          yyerror("divide by zero");
        } else {
          $$ = $1 / $3; printf("reduce to term from %f / %f\n", $1, $3);
        }
      }
    | factor {
        $$ = $1; printf("reduce to term from factor:%f\n", $1);
      }
    ;
```

```
statement_list -> statement |
                  statement_list statement

Statement -> ID '=' term |
             term

term -> term '*' factor |
        term '/' factor |
        factor

factor -> '(' term ')' |
          '-' factor |
          DOUBLE|
          ID
```

Yacc example

```
factor: '(' term ')' {
        $$ = $2; printf("reduce to factor from parentheses\n");
       }
     | '-' factor {
        $$ = -$2; printf("reduce to factor from minus sign\n");
       }
     | DOUBLE {
        $$ = $1; printf("reduce to factor from DOUBLE:%f\n", $1)
       }
     | ID {
        $$ = $1->value; printf("reduce to factor from ID:%s(%f)\n", $1->id, $1->value);
       }
     ;

%%

int yyerror(char *errmsg) {
  fprintf(stderr, "%s", errmsg);
  return 0;
}

int main() {
  yyparse();
  fflush(NULL);
  return 0;
}
```

```
statement_list -> statement |
                  statement_list statement

Statement -> ID '=' term |
             term

term -> term '*' factor |
        term '/' factor |
        factor

factor -> '(' term ')' |
          '-' factor |
          DOUBLE|
          ID
```

Precedence/Association

- Consider two cases:
  - (1) 1-2-3 (association)
  - (2) 1-2*3 (precedence)
- Grammar:
  - expr: expr '-' expr
    | expr '*' expr
    | expr '<' expr
    | '(' expr ')' ;
- (1) 1-2-3 = (1-2)-3 or 1-(2-3) ?
  - Define '-' operator is left association.
- (2) 1-2*3 = 1-(2*3)
  - Define '*' operator to be precedent to '-' operator.

Precedence/Association

- Write in Yacc definition section.
  - % left ' + ' ' - '
  - % left ' * ' ' / '
  - % noassoc UMINUS

low precedence

high precedence

  - % left means left association, % right means right association.

```
expr    : expr ‘+’ expr          { $$ = $1 + $3; }
        | expr ‘-’ expr          { $$ = $1 - $3; }
        | expr ‘*’ expr          { $$ = $1 * $3; }
        | expr ‘/’ expr          {
                                        if($3==0)
                                                yyerror(“divide 0”);
                                        else
                                                $$ = $1 / $3;
                                        }
        | ‘-’ expr %prec UMINUS {$$ = -$2; }
```

Shift/Reduce conflicts

- Shift/Reduce conflicts:
  - Occurs when a grammar is written in such a way that a decision between shifting and reducing can not be made.
  - ex: IF-ELSE ambiguous.
- To resolve this conflict, Yacc will choose to shift.

Reduce/Reduce conflicts

- Reduce/Reduce conflicts:
  - start : expr | stmt ;
  - expr : CONSTANT;
  - stmt : CONSTANT;
- Yacc(Bison) resolves the conflict by using the rule that occurs earlier in the grammar. **NOT GOOD!!**
- Modify grammar to eliminate them.

Error messages

- Bad error message:
  - Syntax error.

  Compiler should give programmers a good advice!

- It is better to track the line number like:

```
test.y
…
%%
…
%%
int yyerror(char *s) {
  fprintf(stderr, "line %d: %s\n:", lineno, s);
}
…
```

Yacc declaration

%start

Specify the grammar's start symbol.

%union

Declare the collection of data types that semantic values may have.

%token

Declare a terminal symbol (token type name) with no precedence or associativity specified.

%type

Declare the type of semantic values for a nonterminal symbol

Yacc declaration

%right

Declare a terminal symbol (token type name) that is right-associative

%left

Declare a terminal symbol (token type name) that is left-associative

%nonassoc

Declare a terminal symbol (token type name) that is non-associative
(using it in a way that would be associative is a syntax error,
ex: **x operand y operand z** is syntax error.)

Homework2

Homework2

- 請在自己的家目錄下建立 Assignment2 資料夾
  - $ mkdir Assignment2
  - $ cd Assignment2
- 複製作業一的 Lex 檔案到 Assignment2，並更改檔名
  - $ cp 學號_hw1.l ./
  - $ mv 學號_hw1.l 學號_hw2.l
- 建立作業二的 Yacc 檔案
  - $ vim 學號_hw2.y
- 所有作業放置在 Assignment2 資料夾底下即可，助教評分會直接使用資料夾內的檔案
- 檔案命名格式錯誤：**扣分!**

Homework2

- 三次作業都是使用這個 testfile.c 來做範例

```
/* This is compiler design homework. */

/* To code,
   or not to code. */

// function declaration
int sub(int x, int y);

int main(){

  // variables
  int a;
  int b=1;
  double c=0;
  char d='x';

  // statements
  a = 10/2;
  c = (b+3)*4-5;
  b = sub(10,8);

  return a;

}

// function
int sub(int x, int y){
  return x-y;
}
```

Homework2

- 作業目的:
  - 把 Token 依照你撰寫的文法一步一步慢慢組合，最終可變成 start symbol
- 輸出結果:
  - 每碰到一個文法導致 symbol 轉變或是 reduce，就印出狀況
  - 例如：
    - input: 12 + 36
    - 印出結果
      - NUM -> value
      - value '+' NUM -> expr
  - Lex 原本印出的東西不需要更動，只需要在 Yacc 的部分印出狀況
  - 輸出格式: □為空格，Yacc輸出範例: value □ '+' □ NUM □ -> □ expr
    - **(違者扣分)**

Homework2

- Lex program 需要修改處:
  - (1) Lex 要回傳 Yacc 定義的 Token 給 Yacc
  - (2) Lex 不需要 main() 函式，否則會跟 Yacc 發生衝突
  - (3) 要 include Yacc 產出的 y.tab.h 檔案
- 提示:
  - 不要一開始就使用 testfile.c 做輸入，可先從比較小的程式片段開始，再慢慢擴大

Homework2

- 作業編譯流程
  - 此次使用 Berkerly Yacc(byacc)
  - $ byacc –d 學號_hw2.y
  - $ flex 學號_hw2.l
  - $ gcc lex.yy.c y.tab.c –lfl
  - $./a.out < testfile.c
- 助教會使用這樣的流程來編譯你們的作業，若按照這樣的指令無法產出檔案，編譯以及執行，

一律扣分!

Homework2

- 配分
- 10% function declaration
- 16% variables
- 16% statements
- 16% function call
- 10% return
- 16% function
- 16% program
  - start symbol: program

```c
/* This is compiler design homework. */

/* To code,
   or not to code. */

// function declaration
int sub(int x, int y);

int main(){

  // variables
  int a;
  int b=1;
  double c=0;
  char d='x';

  // statements
  a = 10/2;
  c = (b+3)*4-5;
  b = sub(10,8);

  return a;

}

// function
int sub(int x, int y){
  return x-y;
}
```

Homework2

- 作業繳交
  - 請在 Assignment2 資料夾中放入
    - 學號_hw2.l
    - 學號_hw2.y
  - 未完成全部項目的同學請額外附上 Readme.txt 檔案
    - 1.能夠成功 parse 的項目
    - 2.為何不能解析剩餘程式
    - 3.嘗試過的方法，遭遇的狀況
- 檔案命名錯誤，**扣分!**

Homework2

- Deadline：2015/05/21 (四) 23:59

Homework2

- 參考書籍
- lex & yacc, 2nd Edition
  - by John R.Levine, Tony Mason & Doug Brown
  - O'Reilly
  - ISBN: 1-56592-000-7

Homework2

Any question?