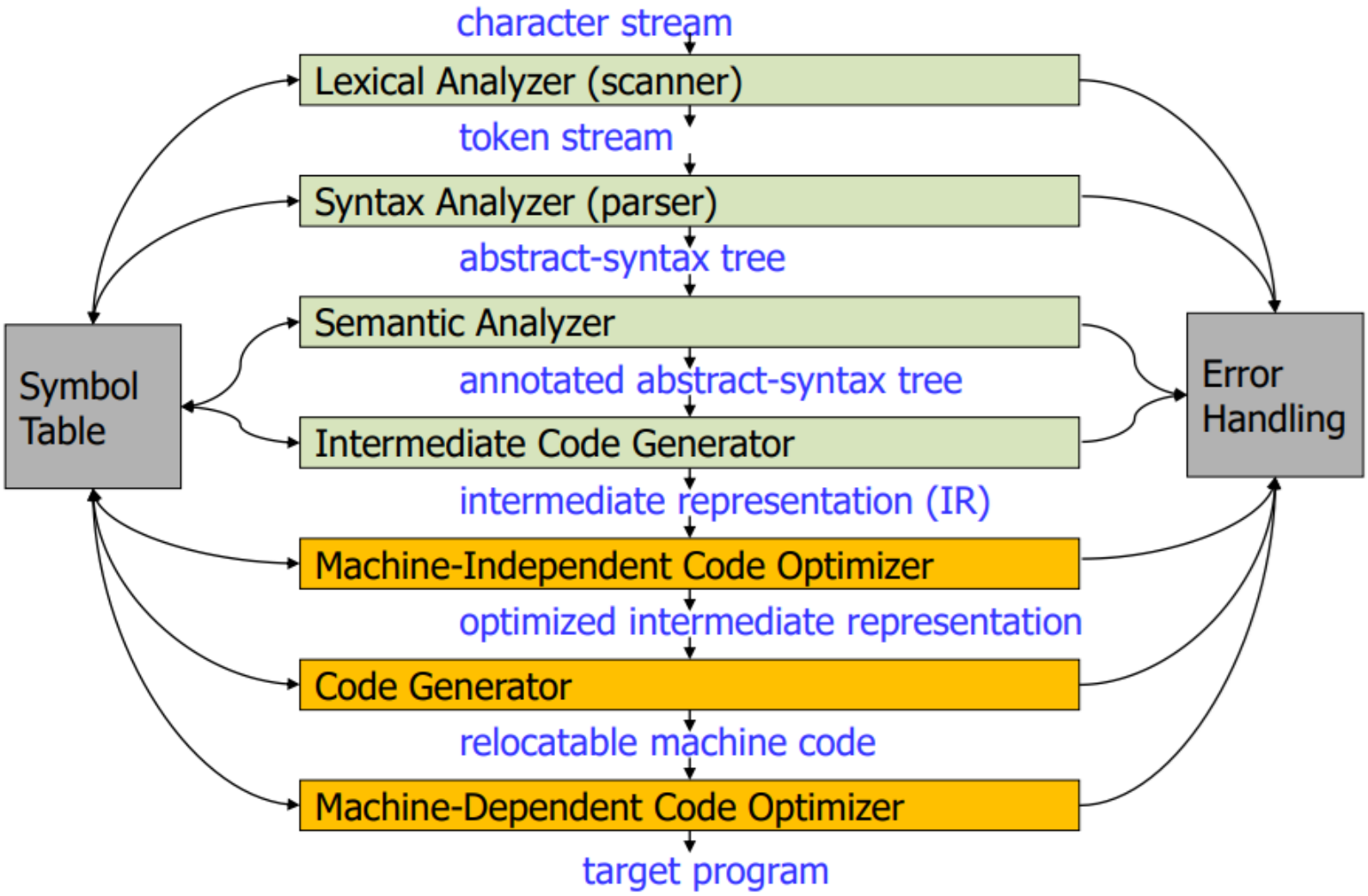
YACC Tutorial

助教 鄭子瀚

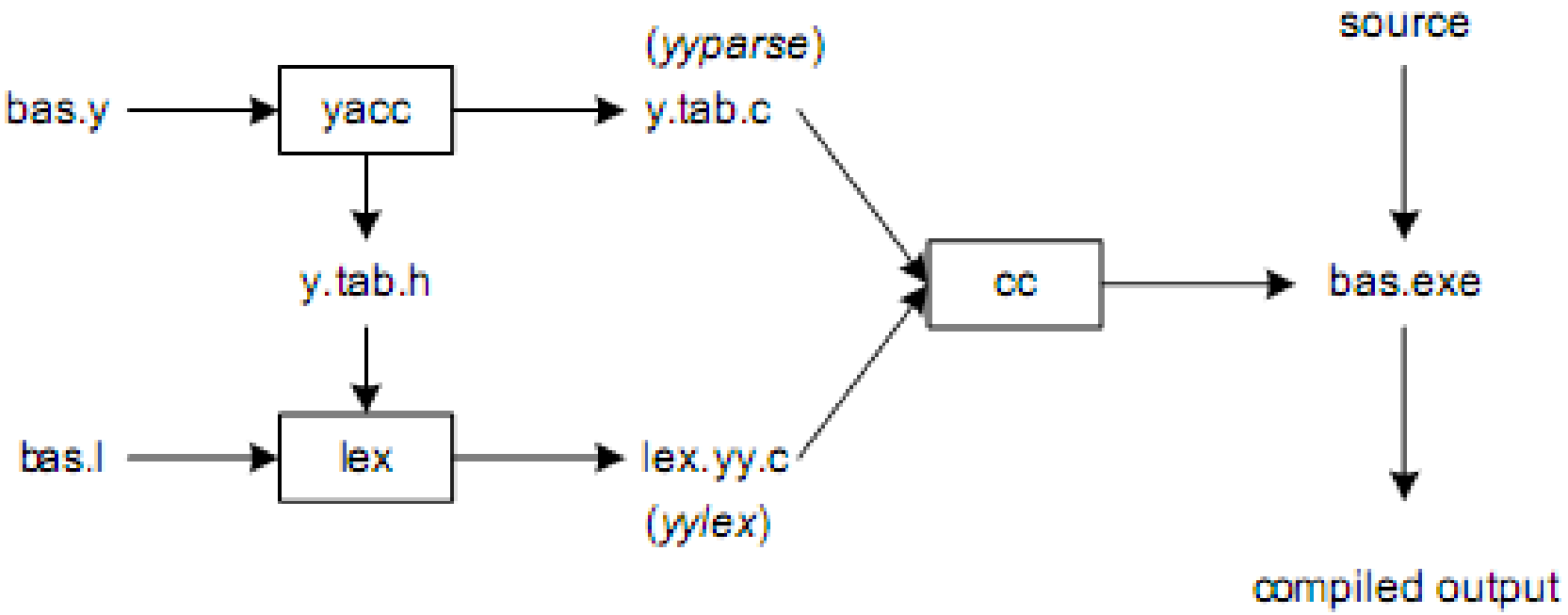
The structure of a compiler



YACC的工作

- YACC會把input當作 a sequence of tokens
  - 一個以上連續的token可以被表示成一個grammar(語法)
- YACC的目的是檢查語法（grammer）是否合法
- Lex只是YACC的一個routine
  - 負責回傳token給YACC

YACC的工作


(yyparse)
source
bas.y
yacc
y.tab.c
y.tab.h
cc
bas.exe
bas.l
lex
lex.yy.c
(yylex)
compiled output

YACC如何表示語法

- 假設現在要做一個簡單的計算機的parser
- 設計語法，其中NUMBER為lex抓到的token:
  - expression -> NUMBER
  - expression -> expression + NUMBER
  - expression -> expression – NUMBER
- 同一個LHS可以合併在一起，再用| 隔開每個RHS
- 以上語法，在YACC會被表示成
  - expression : NUMBER
    | expression + NUMBER
    | expression – NUMBER

LHS或RHS

- 範例： expression -> expression + term
  - LHS: expression
  - RHS: expression + term
- 範例： Article -> a | the
  - LHS: Article
  - RHS: a 或 the（豎線表示選擇）

YACC格式

- 總共分成三個部分
- definition

  %%

  grammars

  %%

  user code
- 每個部分以%%區隔開來

第一部分：Definition

```
%{
#include <stdio.h>
int yylex();
double ans = 0;
void yyerror(const char* message) {
    printf("Invaild format\n");
};
%}
```

```
%union {
    float   floatVal;
    int     intVal;
}
%type <floatVal> NUMBER
%type <floatVal> expression term
                 factor group
%token PLUS MINUS MUL DIV
%token LP RP
%token NUMBER NEWLINE
```

第一部分：Definition

- 在第一部分中，主要分成定義，以及聲明部分
- 例如：定義會在%{ }%之間，而聲明會像是
  - %type <floatVal> NUMBER

第一部分：Definition

- %union：定義了一個可以儲存多種數據類型的集合。
- %type：指定非終結符的數據類型。上面的範例中，expression、term、factor、group 使用 floatVal 從中取值。
- %token：定義了各種token，如 PLUS、MINUS 等，它們代表輸入的運算符。

第二部分：Grammars

```
%%
lines  : /* empty (epsilon) */
       | lines expression NEWLINE
         {printf("%lf\n", $2);}
       ;
expression : term { $$ = $1; }
       | expression PLUS term { $$ = $1 + $3; }
       | expression MINUS term { $$ = $1 - $3; }
       ;
term:  factor {  $$ = $1; }
       | term MUL factor { $$ = $1 * $3; }
       | term DIV factor { $$ = $1 / $3; }
       ;
```

第二部分： Grammars

```
factor:  NUMBER { $$ = $1;}
        | group {$$ = $1; }
        ;
group:  LP expression RP { $$ = $2; }
        ;
%%
```

第二部分：Grammars

- 1. expression 規則:
  - expression : term { $$ = $1; }

    | expression PLUS term { $$ = $1 + $3; }

    | expression MINUS term { $$ = $1 - $3; }

    $1=expression
    $3=term

  - term：若僅為一個 term 時，直接賦值給 expression。
  - 其餘匹配到加法或者減法規則時，$ $被賦予該值。

第二部分：Grammars

- 2. term 規則:
  - term: factor { $$ = $1; }

    | term MUL factor { $$ = $1 * $3; }

    | term DIV factor { $$ = $1 / $3; }

    ;

  - factor：若僅為一個 factor 時，直接賦值。
  - 其餘匹配乘法或者除法，再賦值給 $ $ 。

第二部分：Grammars

- 上述expression 和 term 的功用好像差不多？

- 數學優先級問題:

Operator	Precedence
Not, unary +, unary −, @, **	Highest (first)
*, /, div, mod, and, shl, shr, as, <<, >>	Second
+, −, or, xor, ><	Third
=, <>, <, >, <=, >=, in, is	Lowest (Last)

第二部分：Grammars

- 3. lines 規則:
  - lines : /* empty (epsilon) */ | lines expression NEWLINE
    {printf("%lf\n", $2);} ;
  - 空的行 (epsilon)：表示 lines 可以是空的，允許檔案或輸入以空行開始或結束。
  - lines expression NEWLINE：當解析到 expression 後跟一個換行符時，會觸發 {printf("%lf\n", $2);} ，這裡 $2 表示 expression 的計算結果，並將結果輸出。

第二部分： Grammars

```
expression : | expression   PLUS   term      { $$ = $1 + $3; }
```

$$ $1 $2 $3

```
expression : | expression  {…}  PLUS   term      { $$ = $1 + $4; }
```

$$ $1 $2 $3 $4

第三部分：User Code

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

Lex內容 -- Definition

```
%{
#include "y.tab.h"
#include <stdio.h>
%}
Digit [0-9]+
%%
```

Lex內容 -- Rules

```
{Digit}          { sscanf(yytext, "%f", &yylval.floatVal); return NUMBER;}
\+               {return PLUS;}
\-               {return MINUS;} { /*遇到減號時，返回 MINUS標記 。
                                   */ }
\*               {return MUL;}
\/               {return DIV;}
\(               {return LP;}
\)               {return RP;}
\n               {return NEWLINE;}
.                {return yytext[0];}        { /*其他，返回該字符的 ASCII
值*/ }
%%
```

Lex內容 – User Code

```
int yywrap(){
    return 1;
}
```

編譯流程

- 請先安裝flex和bison
  - sudo apt-get install flex
  - sudo apt-get install bison
- 編譯cau.y　(產生 y.tab.c 及 y.tab.h)
  - bison –y –d cau.y
- 編譯cau.lex　(產生 lex.yy.c)
  - flex cau.l
- 透過gcc產生可執行檔 (產生calc這個執行檔)
  - gcc lex.yy.c y.tab.c -ly –lfl -o calc
- 執行方式
  - ./calc < testfile

執行結果

testfile

```
1  5/2
2  9*3
3  ((3+5)*(8-2))
4  4+3*9-10*8
5
```

```
likems@DESKTOP-6BNKECN:~/yacc$ make
rm -f calc lex.yy.c y.tab.c y.tab.h
bison -y -d calc.y
flex calc.l
gcc lex.yy.c y.tab.c -ly -lfl -o calc
likems@DESKTOP-6BNKECN:~/yacc$ ./calc < testfile
2.500000
27.000000
48.000000
-49.000000
```

Error Recovery

```
51  term: term MUL {strcat(msg, " * ");} factor {
52          /* printf("6\n"); */
53          $$ = $1 * $4;
54      }
55      | term DIV {strcat(msg, " / ");} factor {
56          /* printf("7\n"); */
57          $$ = $1 / $4;
58      }
59      | factor {
60          /* printf("8\n"); */
61          $$ = $1;
62      }
63      | error NUMBER { /* Error happened, discard token until it find NUMBER. */
64          /* printf("9\n"); */
65          yyerrok;     /* Error recovery. */
66      }
67      ;
```

Error Recovery

testfile_err

```
1  3++9
2  5/2
3  9*3
4  3+5
5  4**6
6  5+***6+*6
7  4+3*9-10*8
8
```

```
likems@DESKTOP-6BNKECN:~/yacc_calc_err$ ./calc < testfile_err
syntax error at line 1
line 2: 5 / 2    (ans = 2.500000)
line 3: 9 * 3    (ans = 27.000000)
line 4: 3 + 5    (ans = 8.000000)
syntax error at line 5
syntax error at line 6
syntax error at line 6
line 7: 4 + 3 * 9 - 10 * 8       (ans = -49.000000)
likems@DESKTOP-6BNKECN:~/yacc_calc_err$
```

編譯流程

- 在example中，助教有幫大家寫好makefile，如下

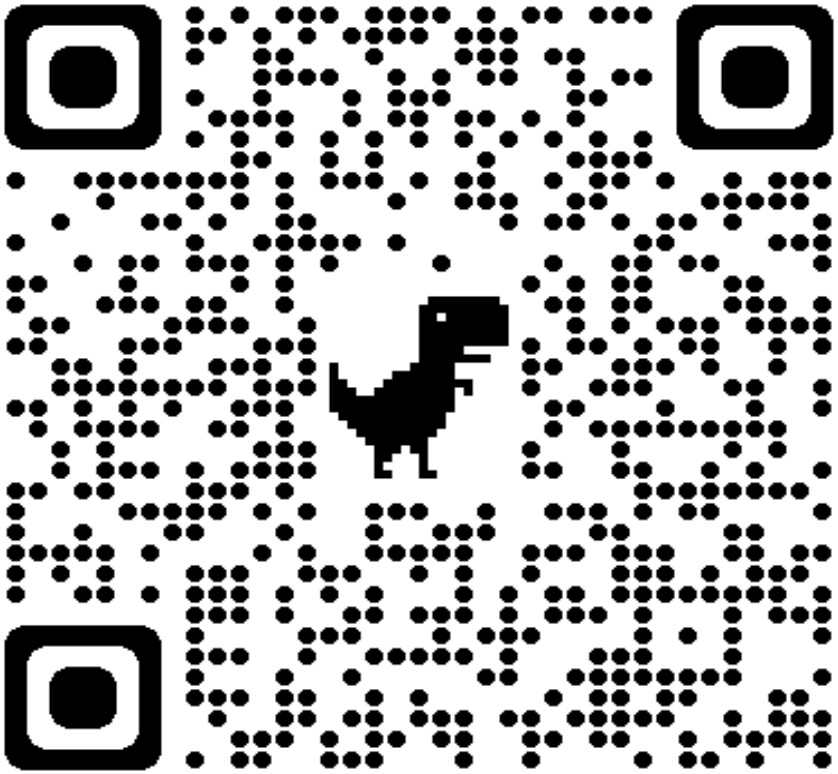
```
all:  clean y.tab.c lex.yy.c
    gcc lex.yy.c y.tab.c -ly –lfl –o calc
y.tab.c:
    bison -y -d cau.y
lex.yy.c:
    flex cau.l
clean:
    rm -f calc lex.yy.c y.tab.c y.tab.h
```

- 執行「make all」即可編譯產生「calc」

作業繳交注意事項

- **due: 5/19 11:59p.m.**
- YACC的設計要比Lex要相對複雜，因此建議早點開始撰寫。
- 程式Demo環境是Ubuntu 22.04，因此請保證你們的程式碼能夠在Ubuntu上面編譯執行
- 作業說明有提供input file，可自行驗證
- 請準時繳交作業，作業遲交一天打七折
- 請把作業包成一個壓縮包，上傳至網大，檔名命為 「學號_hw2」，學號輸錯，此項作業分數-10，沒輸學號分數-50，請同學注意
- 作業繳交之後，在繳交截止隔周會安排時間，到EC5023找助教Demo。

預約時段



YACC-28