# 112-2 國立成功大學編譯系統課程

## NCKU Compiler Construction - 2024 Spring

## Homework 2. Syntactic Analysis (Parser)

Deadline: 2024/05/24 23:59:59

Late submissions are not accepted

# 作業規範

- 請不要攻擊我們的伺服器，否則這堂課會直接當掉
- 作業不接受任何形式的補交或遲交，請大家在截止前完成
- 作業截止後，我們會將所有同學的程式碼送到我們的比對系統中，如有發生作業抄襲等情形，抄的人與被抄的人學期作業成績 (40 %) 以 0 分計算

# 本學期的作業配分

- 這學期總共有 3 份作業，作業成績佔學期成績 40%
  - Homework 1: 10%
  - Homework 2: 15%
  - Homework 3: 15%

# Homework 1

- 滿分 120, 平均 111 分
- 作業一的成績已經公告在 moodle
- 對成績有疑問的請 4/13 晚上 11 點之前來信
- 逾期之後不再受理作業一的成績問題

# 有人私底下跟我討論了這件事...


# 那我們改名成 C--

# 編譯器的步驟

- 詞法分析 Lexical analysis ( Scanning )
- 語法分析 Syntax analysis ( Parsing )
- 語意分析 Semantic analyzer
- 中間碼生成
- 程式碼優化

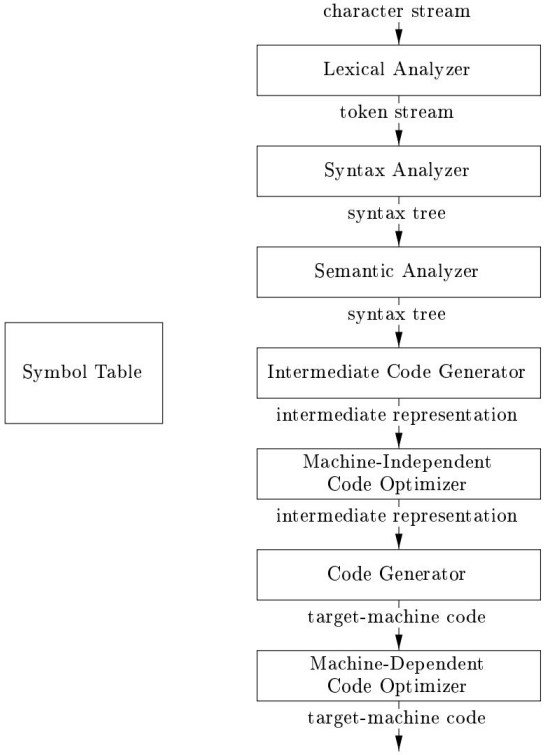
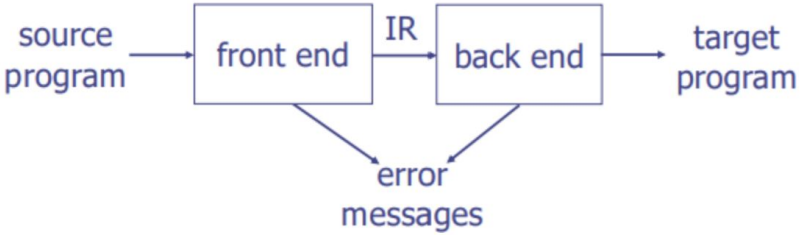



Figure 1.6: Phases of a compiler

# Analysis-Synthesis Model




- 綜合來說，編譯的流程可以分成：
  - 分析 Analysis ( front end )
    - 將程式拆成很多零件
    - 得到中間碼 intermediate representation (IR)
  - 合成 Synthesis ( back end )
    - 利用中間碼組合出目標程式
    - 最佳化 (可做可不做)

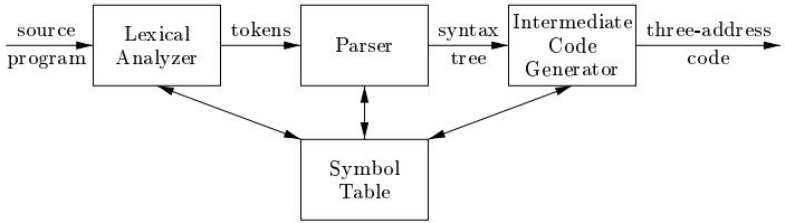
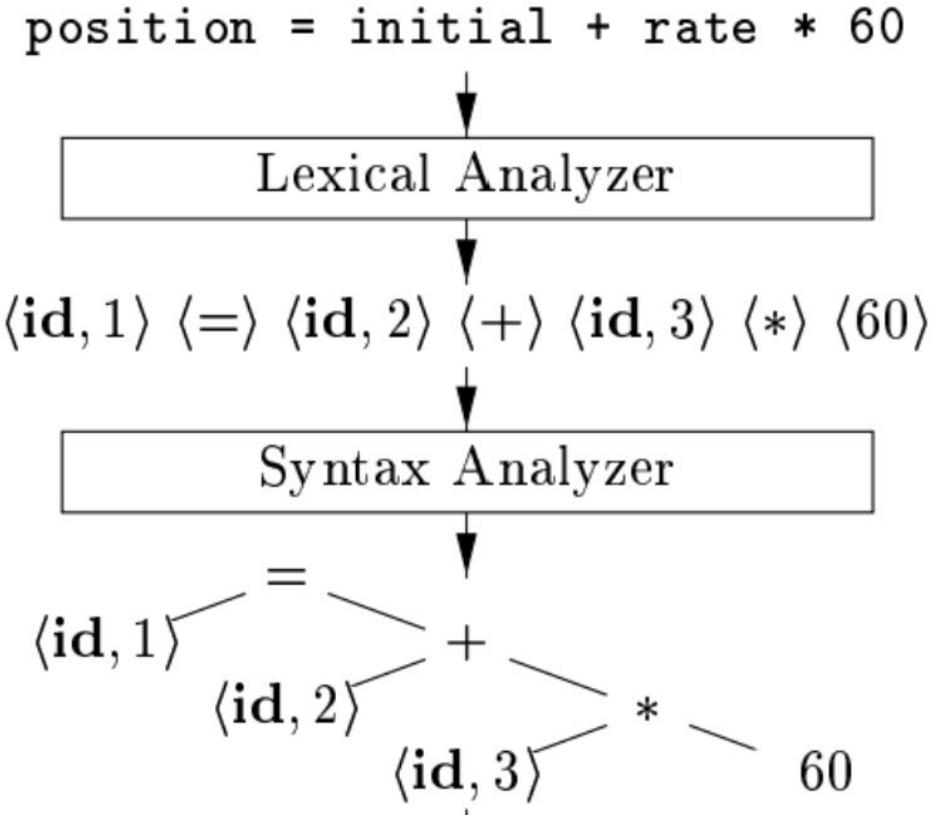



Figure 2.3: A model of a compiler front end

# Analysis

- Linear Analysis (Lexical Analysis)
  - 掃描程式碼，將文字拆解成許多片段 ( Token )
- Hierarchical Analysis (Syntax Analysis) (Homework 2)
  - 將這些 Token 組成文法
- Semantic Analysis
  - 辨識語法錯誤，與型別問題

# Analysis


position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨∗⟩ ⟨60⟩
Syntax Analyzer
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
60

# Syntactic Analysis (Parser)

- 這是實作 Compiler 的第二個步驟!
- 語法分析，就是 check 語法對不對的一個步驟
- 就像是英文有自己的文法，而 Parser 要做的事情就是 check 程式碼的語法是否符合規定

# Syntactic Analysis (Parser)

- 作業一已經將程式碼切成很多 Token
- 接下來我們要把 Token 照順序放入 Parser 解析語法
    - 你可以想像成我們要把這些 Token

      組合成一個可以被 "理解" 的句子
    - 而要創造出一個句子，我們必須要先有語法
    - 所以現在我們要制訂出一套文法標準

# Syntax Definition 語法定義

- 每一個語言都有一個既定的 "規則"
- 這一個規則描述了程式語言在編寫時必須遵守的一些規範
- 我們通常把一個 "規則" 稱之為 文法 ( Grammar )

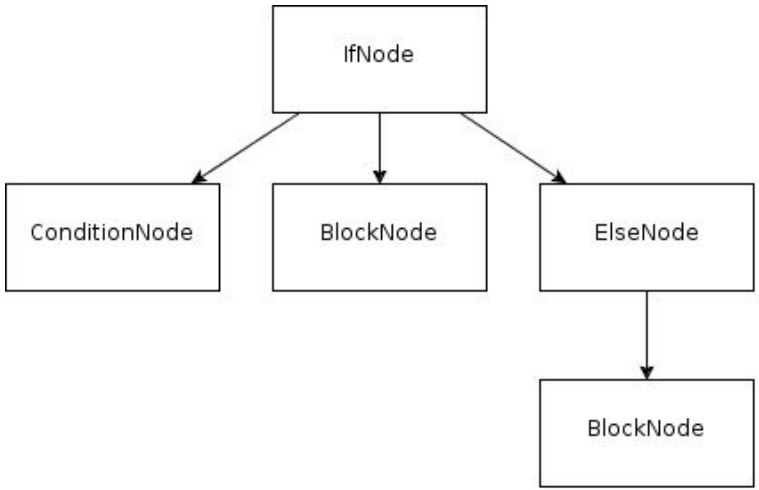
**if** ( expression ) statement **else** statement

# Syntactic Analysis (Parser)

- 我們的目標是：
  - 將 Token 重新以正確的順序 (優先級) 輸出
  - 簡易的判斷所有變數的可視範圍 (Scope)
  - 針對所有的 Scope Level 輸出一個表格，表格內有該等級範圍內的所有變數資訊

# Grammar Design 語法設計

- 這個步驟我們最重要的是要建立出一個 Parser Tree
- 這一個樹要能表示解析所有的語法規則



# Grammar Design

- 建立 Parser Tree 的種類又分成兩種：
  - Top-Down Parser (從根開始往下)
  - Bottom-Up Parser (從葉子開始往上長)
  - 但在這一個作業我們會直接使用別人寫好的工具，這一個工具會幫助我們更快的建立出 Parser Tree

# Grammar Design (Yacc)

- Yet Another Compiler Compiler
- 使用 LALR(1) 的方式做語法分析
- 我們可以用 BNF (巴科斯範式) 的表示法來直接表明我們想要設計的文法
    - 這邊特別科普一下：其實 BNF 就只是 CFG 所衍生出來的一種特殊寫法而已，所以本質上還是以 CFG 為基礎

# Grammar Design (Yacc)

- 設計文法的一開始我們要跟 Lex 做搭配，先定義好所有 token 的屬性
- 這樣你設計文法的時候 Yacc 才會認得這個 token 是誰

```
8   /* Token without return */
9   %token LET MUT NEWLINE
10  %token INT FLOAT BOOL STR
11  %token TRUE FALSE
12  %token GEQ LEQ EQL NEQ LOR LAND
13  %token ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN REM_ASSIGN
14  %token IF ELSE FOR WHILE LOOP
```

# Grammar Design (Yacc)

- 針對有 value 的 token 我們要背著他的屬性
- 所有 token 的型態在一開始 %union 要先定義好

```
%union {
    int i_val;
    float f_val;
    char *s_val;
}
```

```
/* Token with return, which need to sepcify type */
%token <i_val> INT_LIT
%token <f_val> FLOAT_LIT
%token <s_val> STRING_LIT
```

# Grammar Design (Yacc)

- 如何設計文法？(BNF 格式)
- 下列文法可解析：
    - int apple = 10;
    - int Sumikko_Gurashi;

```
Sample // 宣告變數
    : INT ID '=' INT_LIT ';'
    | INT ID ';'
```

# Grammar Design (Yacc)

- 我們解析文法的時候要搭配語意動作
- 輸出我們當前解析到的資訊，確保我們解析過程正確
- 只要在符號的右方使用大括號即可
- 當解析到 int 時就會執行語意動作

```
1  Sample // 宣告變數
2      : INT { printf("%d\n","INT") } ID '=' INT_LIT ';'
3      | INT ID ';'
```

# 語法制導翻譯方案 Syntax-Directed Translation Scheme

- 語意動作可以讓我們決定該動作執行的時間 (順序)
- 遍歷語法樹時，到達語意動作產生的節點才會執行該動作

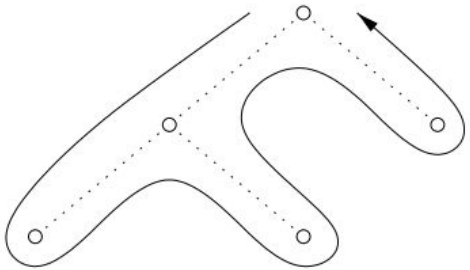


Figure 2.12: Example of a depth-first traversal of a tree

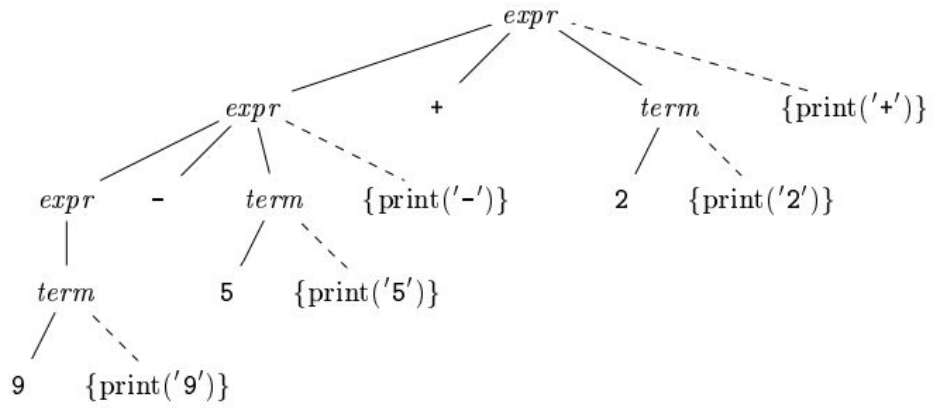



Figure 2.14: Actions translating 9-5+2 into 95-2+

# Grammar Design (Yacc)

- 其餘更詳細的語法請參閱課程簡報：D
- 接下來我們講解我們希望你的 Yacc 可以輸出什麼

# Grammar Design (Yacc) - Output

- 將 Token 重組後的順序與相關資訊
- Symbol Table (等等會講為什麼正常來說需要這個東西)

```
> Create symbol table (scope level 0)
func: main
> Insert `main` (addr: -1) to scope level 0
> Create symbol table (scope level 1)
> Insert `argv` (addr: 0) to scope level 1
STR_LIT "Hello World"
IDENT (name=endl, address=-1)
cout string string
INT_LIT 0
RETURN

> Dump symbol table (scope level: 1)
Index     Name                Type      Addr      Lineno    Func_sig
0         argv                string    0         1         -

> Dump symbol table (scope level: 0)
Index     Name                Type      Addr      Lineno    Func_sig
0         main                function  -1        1         ([Ljava/lang/String;)I
Total lines: 4
```

# Grammar Design (Yacc) - Output

- 將 Token 分析意思後，以正確的順序輸出

```
int main(string argv[]) {
    cout << ( 3 - 4 * (5 + -8) - 10 / 7) << endl;
    cout << ( 3 - 4 * (5 + -8) - 10 / 7 > -4 % 3 || !true && !!false)
    cout << ( 3.0 - 4.0 * (5.0 + -8.0) - 10.0 / 7.0) << endl;
    cout << 3.0 - 4.0 * (5.0 + -8.0) - 10.0 / 7.0 > -4.0 || !true &&
    return 0;
}
```

```
INT_LIT 3
INT_LIT 4
INT_LIT 5
INT_LIT 8
NEG
ADD
MUL
SUB
INT_LIT 10
INT_LIT 7
DIV
SUB
```

# Grammar Design (Yacc) - Output

- 各個符號的優先級可以參考維基百科上的
- 與我們平時寫的 C/C++ 的優先級一樣
- 或是參考我們的測資也可以！

# 輸出：可視範圍與生命週期

```
main() {
    int a = 1;                                   B1
    int b = 1;
    {
        int b = 2;                               B2
        {
            int a = 3;                           B3
            cout << a << b;
        }
        {
            int b = 4;                           B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Figure 1.10: Blocks in a C++ program

| DECLARATION | SCOPE |
|---|---|
| `int a = 1;` | $B_1 - B_3$ |
| `int b = 1;` | $B_1 - B_2$ |
| `int b = 2;` | $B_2 - B_4$ |
| `int a = 3;` | $B_3$ |
| `int b = 4;` | $B_4$ |

Figure 1.11: Scopes of declarations in Example 1.6

# Grammar Design (Yacc) - Output

- Create symbol table: 當遇到新的 Scope 時觸發
- 需輸出新的 scope level

```
int main(string argv[]) {
    cout << "Hello World" << endl;
    return 0;
}
```

```
> Create symbol table (scope level 0)
func: main
> Insert `main` (addr: -1) to scope level 0
> Create symbol table (scope level 1)
> Insert `argv` (addr: 0) to scope level 1
STR_LIT "Hello World"
IDENT (name=endl, address=-1)
cout string string
INT_LIT 0
RETURN

> Dump symbol table (scope level: 1)
Index     Name                Type      Addr      Lineno    Func_sig
0         argv                string    0         1         -

> Dump symbol table (scope level: 0)
Index     Name                Type      Addr      Lineno    Func_sig
0         main                function  -1        1         ([Ljava/lang/String;)I
Total lines: 4
```

# Grammar Design (Yacc) - Output

- Insert:
- 遇到變數時，輸出他被 Insert 到的 scope 與 address

```
int main(string argv[]) {
    cout << "Hello World" << endl;
    return 0;
}
```

```
> Create symbol table (scope level 0)
func: main
> Insert `main` (addr: -1) to scope level 0
> Create symbol table (scope level 1)
> Insert `argv` (addr: 0) to scope level 1
STR_LIT "Hello World"
IDENT (name=endl, address=-1)
cout string string
INT_LIT 0
RETURN

> Dump symbol table (scope level: 1)
Index     Name                Type      Addr      Lineno    Func_sig
0         argv                string    0         1         -

> Dump symbol table (scope level: 0)
Index     Name                Type      Addr      Lineno    Func_sig
0         main                function  -1        1         ([Ljava/lang/String;)I
Total lines: 4
```

# Grammar Design (Yacc) - Output

- Dump symbol table
- Scope level 從高到低輸出，每一個 scope 內的所有變數名稱，型態，行數
- 如果是 Function 要輸出他的參數是什麼

```
int main(string argv[]) {
    cout << "Hello World" << endl;
    return 0;
}
```

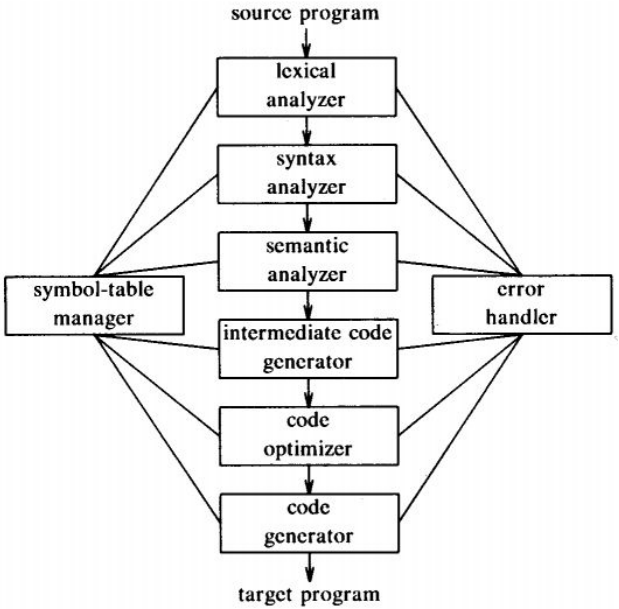
```
> Create symbol table (scope level 0)
func: main
> Insert `main` (addr: -1) to scope level 0
> Create symbol table (scope level 1)
> Insert `argv` (addr: 0) to scope level 1
STR_LIT "Hello World"
IDENT (name=endl, address=-1)
cout string string
INT_LIT 0
RETURN

> Dump symbol table (scope level: 1)
Index     Name                Type      Addr      Lineno    Func_sig
0         argv                string    0         1         -

> Dump symbol table (scope level: 0)
Index     Name                Type      Addr      Lineno    Func_sig
0         main                function  -1        1         ([Ljava/lang/String;)I
Total lines: 4
```

# Grammar Design (Yacc) - Output

- 建議寫的時候參考解答輸出，更加了解每一個測試資料要求輸出的東西！
- 每一個測試資料確認完輸出順序後再動手會比較好 ><

# 為什麼需要 Symbol Table？

- 整個編譯的過程都必須跟符號表（Symbol table）溝通



# Homework 2 作業上傳

- 請先 ssh 到 140.116.154.66 (非成大網路者，請使用 VPN)
- 接著在你自己的資料夾中，clone 下方的 GitHub repo
- https://github.com/ColtenOuO/2024-Spring-NCKU-CompilerHW2/tree/main
- 進到 repo 資料夾後開始編寫 compiler.y

# Homework 2 作業上傳

- 你的 compiler.l, compiler.y, Makefile 必須要放置在 ~/home/你的學號/2024-Spring-NCKU-CompilerHW2
- 這邊聲明一下，我們指的 clone 是指使用 git 指令將這一個 repo clone 下來，不要下載成 zip 後解壓縮

# Homework 2 作業上傳

- 想先在自己系統寫的人請使用以下指令安裝環境
- sudo apt install flex bison

# Homework 2: Grading

- 本次作業滿分為 120 分
- 評分標準與上一個作業相同

# Homework 2: Subtasks

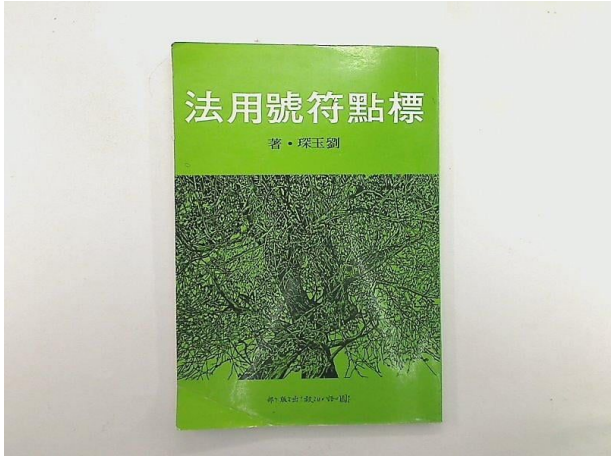
- subtask 01: helloworld (8 %)
- subtask 02: 註解 (8 %)
- subtask 03: 運算優先順序 (8 %)
- subtask 04: 運算符號 (8 %)
- subtask 05: 強制轉型 (8 %)
- subtask 06: 條件判斷 (8 %)
- subtask 07: while 迴圈 (8 %)

# Homework 2: Subtasks

- subtask 08: for (8 %)
- subtask 09: 函式 (8 %)
- subtask 10: 一維陣列 (8 %)
- subtask 11: auto (8 %)
- subtask 12: 多層迴圈 + if (10 %)
- subtask 13: 二維陣列 (8 %)
- subtask 14: 綜合版測試資料 (14 %)

# 作業提示: 注意模稜兩可的情況

- 此現象表示某一個文法可以被解讀成兩個以上的意思
- 就像是我們平時講的句子加上適當的標點符號可能會有很多不一樣的意思
  - 下雨天留客天留我不留 =>
    - 下雨天留客，天留我不留。
    - 下雨天留客，天留我？不留。
    - 下雨天留客，天留我不？留。



# 模稜兩可 Ambiguous

- 同一個文法具有兩個以上的文法樹 (Parse tree)

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

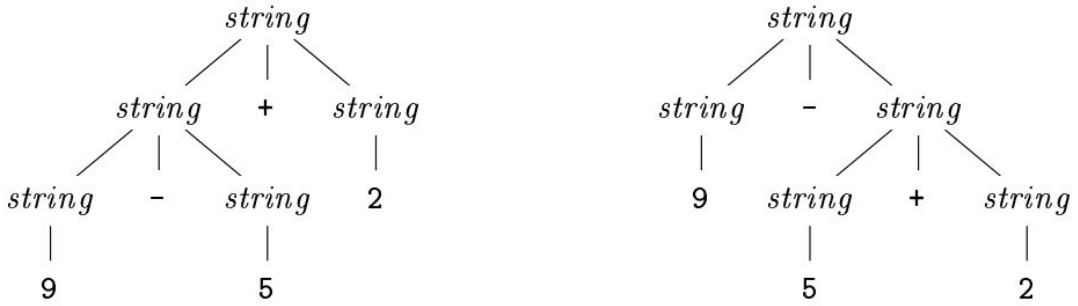



Figure 2.6: Two parse trees for 9-5+2

# 模稜兩可 Ambiguous

- 左圖可當作 ( 9 - 5 ) + 2，右圖可當作 9 - ( 5 + 2 )

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

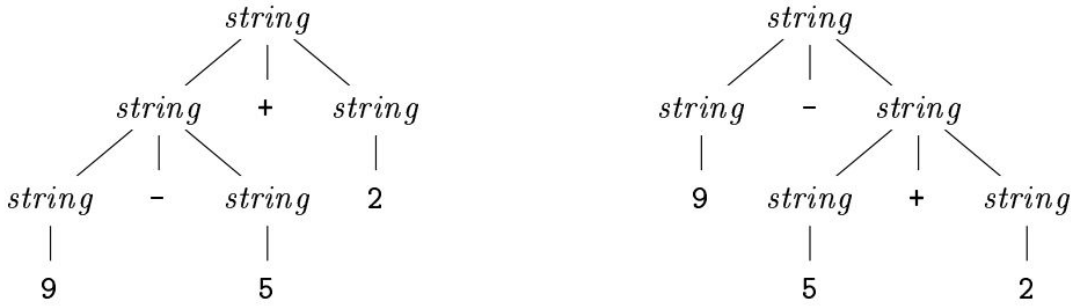



Figure 2.6: Two parse trees for 9-5+2

# 運算符號的結合性

- 依照數學的慣例，9 + 5 + 2 等價於 ( 9 + 5 ) + 2
- 當一個句子兩側都有運算符號時，我們需要制定一個性質 結合性
  - 左結合 left-associative
  - 右結合 right-associative
- 結合性幫助我們明確的知道每一個運算符號該用於誰
  - + 是左結合的，因此我們可以知道 9 + 5 的 + 是用於 9 身上

# 運算符號的結合性

- + - * / 都是左結合的運算符號
- 右結合的運算符號：=、+=、-= and so on…
  - = 我們通常稱為賦予，a = b 我們可以理解成 b 賦予 a
  - 把右邊的 value 給予左邊
  - = 是用於右邊的 b，讓 b 可以賦予給 a

# 作業提示: 注意運算符號的結合性

- 有些符號屬於左結合，有些屬於右結合

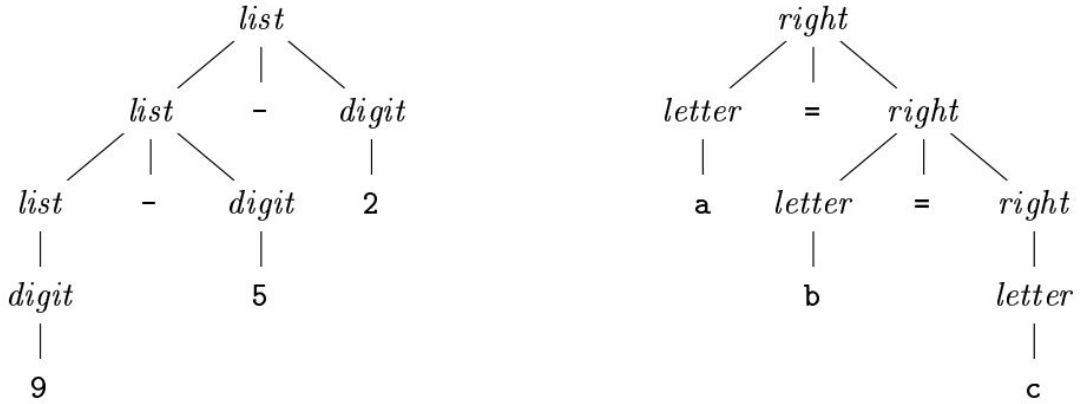



Figure 2.7: Parse trees for left- and right-associative grammars

# 作業提示：有時候光依靠結合性無法解決問題

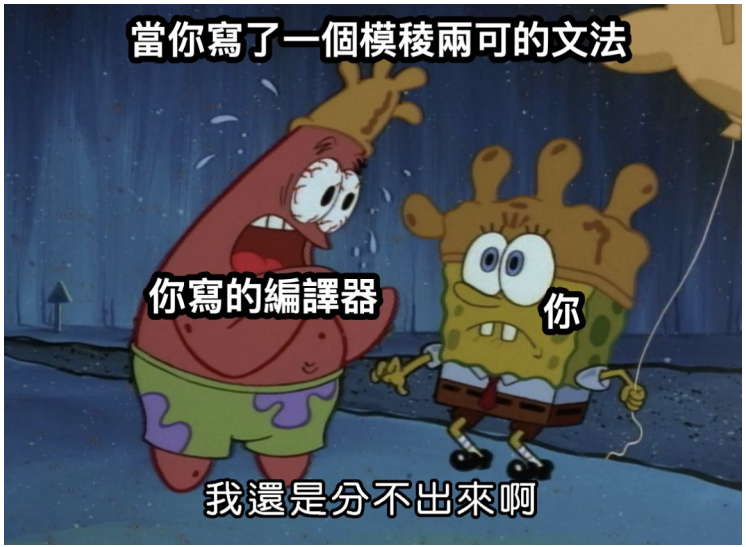
- 當一個文法具有許多運算符號時依舊無法解決模稜兩可的問題
  - 結合性只能解決同一個運算符號重複出現造成的模稜兩可問題
- 因此我們需要再處理不同運算符號之間的優先順序來解決問題
  - 我們可以透過改寫文法來解決這樣的問題
  - 這是最直接也最方便的方法
- 總結：設計一個好的文法是很重要的！

# 作業提示：Symbol Table 善用一些資料結構

- 你「可能」會用到：
  - LinkedList
  - hash
- 也許他們可以讓你變輕鬆

# Homework 2: Challenge Subtask

- 讓你的 Parser 不會發生 Shift-Reduce Conflict
  - 設計文法必須小心，避免發生模稜兩可



# Homework 2: 提醒

- 許多同學作業一切 token 的時候是有點硬做，把每一個變數的名字都獨立成一個 token，這樣其實非常不好！
- 你的 Compiler 會只能辨識那些變數名字！！！
- 請善用正則表達式
- 這一次作業我們測資會隨機生成變數名稱

# Homework 2: 請大家少用 remote-ssh

- 請善用 scp 指令，拜託ＱＱ

# Homework 2: 輸出目前只有 5 筆

- 我們會盡快陸續把所有測資的答案補齊！
- 請大家再耐心稍候一下 ><

# Homework 2: 重要時程

- 作業繳交規定截止日：5/24 23:59:59
- 本學期退選截止日期：5/17

# Homework 2: Final

- 最後祝大家作業順利! Good luck & have fun
- 測資是我們熬夜肝好幾個晚上用出來的，可能難免失手，如果有覺得不對勁的地方可以跟我們說 ><
- f74114744@gs.ncku.edu.tw (資訊 115 陳俊安)
- f74114760@gs.ncku.edu.tw (資訊 115 張羿軒)