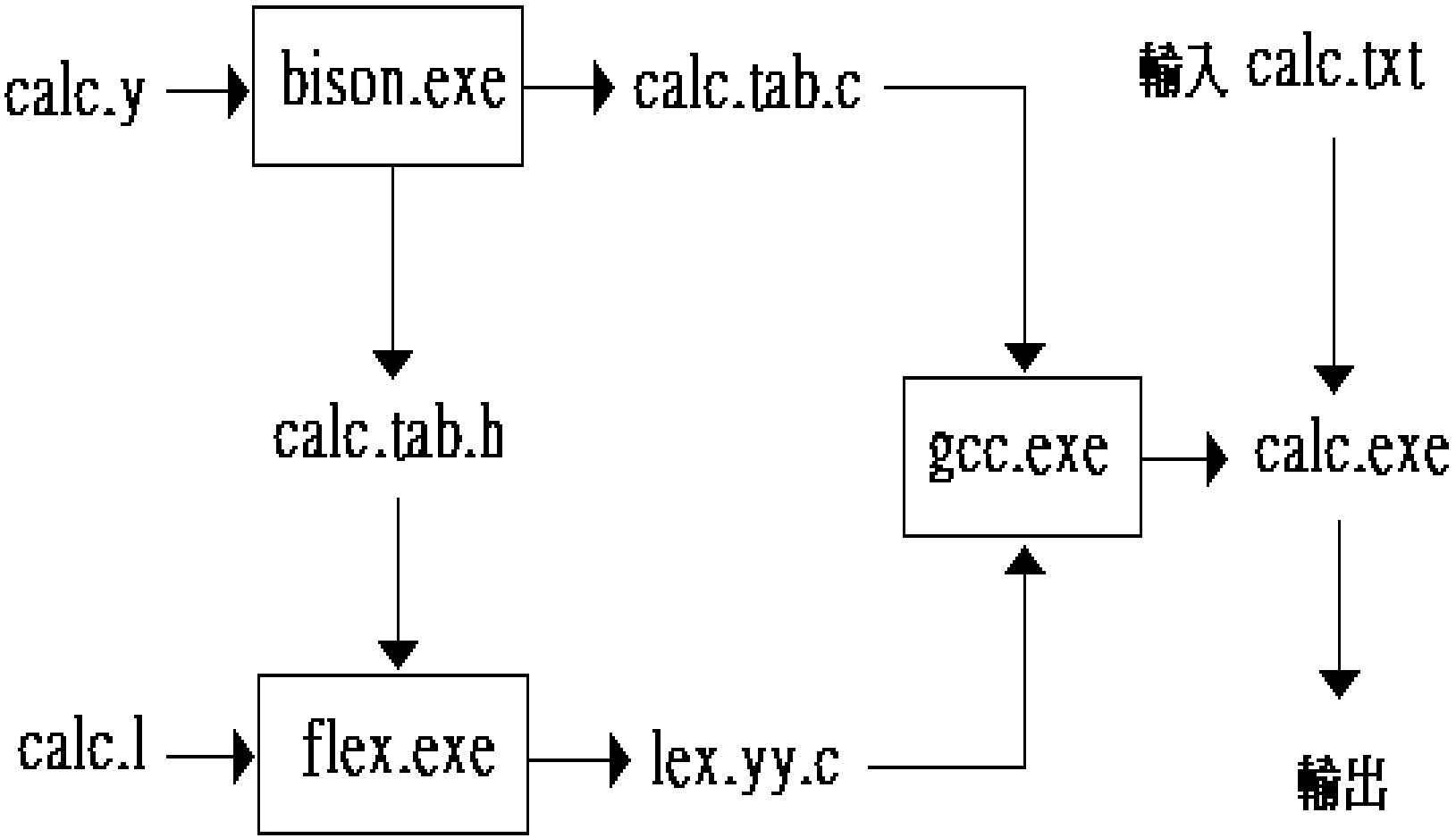
第十五章 計算機編譯器

- 前面兩章分別單獨介紹 flex 及 Bison 的用法，單獨使用 flex 時您要撰寫一段剖析程式碼，單獨使用 Bison 時您要撰寫一段掃描程式碼，來辨識輸入的符記，覺得很麻煩。
-
- 本章要將這兩個軟體，flex 與 Bison，結合起來使用，並以計算機編譯器為例說明 flex 及 Bison 之間的介面，以及其用法。

15.1 flex 及 Bison 之間的介面



- 語彙分析器產生器程式 flex.exe 從規格檔 ***calc.l*** 建立語彙分析器原始程式 ***lex.yy.c***，它也稱為掃描程式。掃描程式 ***lex.yy.c*** 的工作就是要從輸入檔 ***calc.txt*** 輸入字元，匹配指定的樣式，轉成符記給剖析器產生器程式 bison.exe 所產生的剖析器程式 ***calc.tab.c*** 使用。
- 這兩個程式，***lex.yy.c*** 與 ***calc.tab.c***，透過 gcc 編譯器編譯成可執行檔 ***calc.exe***。執行時您要提供一個輸入檔，執行結果會顯示在螢幕上，當然您也可以將輸出轉至指定的本文檔案。
- 執行 bison.exe 命令如下：
- ***bison -d calc.y <Enter>***
- 執行後產生一個 ***calc.tab.c*** 原始程式檔，以及 ***calc.tab.h*** 表頭檔，包含各個符記的整數編號，這個表頭檔必須透過 #define 指引引入至規格檔 ***calc.l*** 裡頭才行。

- 執行 flex.exe 命令如下，產生一個 lex.yy.c 原始程式檔。
- ***flex calc.l <Enter>***
- 執行 gcc.exe 命令如下，將 lex.yy.c 及 calc.tab.c 連結成一個可執行的 calc.exe 程式檔。
- ***gcc lex.yy.c calc.tab.c -o calc.exe <Enter>***
- 為了簡化執行的命令，將這三個命令存入一個 run.bat 批次檔，再加上一個執行的指令，並提供一個輸入檔名參數 %1。
- run.bat 批次檔內容如下：
- ***bison -d calc.y***
- ***flex calc.l***
- ***gcc lex.yy.c calc.tab.c -o calc.exe***
- ***calc.exe %1***

15.2 後置記法計算機專案

- 後置記法計算機請參閱上一章。本小節只是將掃描函式 yylex() 從文法檔案 calc.y 抽出，另外製作一個 calc.l 規格檔後再將它們結合起來而已。
- 本專案存於 ***C:\plone\rpncalc*** 目錄。
- 在 flex 的規格檔 calc.l 中：
- ***#include "calc.tab.h"***
- 這個表頭檔 calc.tab.h 由 Bison 所建立，必須將它引入 flex 規格檔 calc.l 裡頭，否則執行 flex.exe 時會產生錯誤的。

- ***[0-9]+ {***
- ***sscanf(yytext, "%lf", &yylval);***
- ***return NUM;***
- ***}***
- 正規運算式 [0-9]+ 表示樣式最少要有一個阿拉伯數字，其後可以跟著若干個阿拉伯數字，這個符記稱為 NUM，其字串為 yytext，經 C 語言的 sscanf() 函式轉換成倍精確浮點數後存入 yylval 整體變數，這個整體變數是 Bison 的語意（semantic）值。
- ***[0-9]+"."[0-9]****
- 這個正規運算式表示小數點前面最少要有一個阿拉伯數字，小數點後面可以有阿拉伯數字，也可也沒有阿拉伯數字。例如 365.、123.45 均可。

- ***[0-9]*"."[0-9]+***
- 這個正規運算式表示小數點前面可以有阿拉伯數字，也可也沒有阿拉伯數字。但小數點之後一定要有阿拉伯數字。例如 365.25、.123 均可。
- ***[-+*/^n\n]***
- 這個正規運算式表示單一字元的符記，包含 '-'、'+'、'*'、'/'、'^'、'n'、'\n' 等七個。只是單純的將該字元傳回。
- ***.***
- 這個句點「.」樣式表示前面無法匹配的字元它都接受，表示錯誤，對於本專案來講它屬於非法的字元。

- Bison 文法檔 calc.y 如後，其中：
-
- ***#include <stdio.h>***
- ***#define YYSTYPE double***
- ***int yylex(void);***
- ***void yyerror (char const *);***
- ***extern FILE *yyin;***
-
- 這是「前導段落」裡的 C 語言程式碼。
-
- 「***#define YYSTYPE double***」定義巨集 YYSTYPE 為 double 型態。「***extern FILE *yyin;***」宣告 yyin 為外部變數，它定義於 lex.yy.c 原始程式裡頭。

- %token NUM
- 它是前導段落裡的Bison宣告，宣告符記 NUM，其型態為 YYSTYPE。
- ***int main (int argc, char *argv[])***
- ***{***
- ***yyin=fopen("calc.txt","r");***
- ***printf("　輸入資料檔\"calc.txt\"內容如下：\n");***
- ***system("TYPE calc.txt");***
- ***printf("　透過bison的yyparse()逐一剖析如下：\n");***
- ***yyparse();***
- ***return 0;***
- ***}***

- 為了要瞭解 Bison 剖析程式的邏輯，我們要在執行 bison.exe 時加入個「-r itemset」的選項，執行如下：
- bison -r itemset -d calr.y <Enter>
- 這時除了產生 calr.tab.c 及 calr.tab.h 兩個檔案之外，還產生一個額外的檔案 calr.output，它是一個 Bison 內部所使用的除錯檔案，包括下列資料：
- ***1. 終端符號的編號及出現在那一條文法規則。***
- ***2. 非終端符號及出現在那一條文法規則的左手方或右手方。***
- ***3. LALR 自動機。***
- 文法檔案 calr.y 是 calc.y 的簡化版，只選 '+' 與 '-' 的規則而已，為了完整起見還是另給一個檔名 calr.y。

- 【Bison除錯檔calr.output】
-
- Grammar [註]文法
- 0 $accept: input $end [註]第0條文法規則
- 1 input: input line
- 2 | /* empty */
- 3 line: '₩n'
- 4 | expr '₩n'
- 5 expr: NUM
- 6 | expr expr '+'
- 7 | expr expr '-'
-
- Terminals, with rules where they appear [註]符記及規則
- $end (0) 0
- '₩n' (10) 3 4 [註]'₩n'的ASCII碼為10,出現在#3、#4規則
- '+' (43) 6 [註]'+'的ASCII碼為43,出現在#6規則
- '-' (45) 7 [註]'-'的ASCII碼為45,出現在#7規則
- error (256)
- NUM (258) 5 [註]NUM符記編號258,出現在#5規則

- Nonterminals, with rules where they appear
- $accept (7)
- on left: 0
- input (8) [註]input(#8規則)
- on left: 1 2, on right: 0 1 [註]出現在左右手邊規則
- line (9)
- on left: 3 4, on right: 1
- expr (10)
- on left: 5 6 7, on right: 4 6 7
-
- state 0 [註]狀態0
- 0 $accept: . input $end [註]小數點右方為輸入資料流
- 1 input: . input line [註]小數點左方為堆疊
- 2 | .
- $default reduce using rule 2 (input) [註]用#2簡化
- input go to state 1 [註]若input跳至狀態1
- state 1 [註]狀態1
- 0 $accept: input . $end
- 1 input: input . line
- 3 line: . '₩n'
- 4 | . expr '₩n'
- 5 expr: . NUM
- 6 | . expr expr '+'
- 7 | . expr expr '-'

15.3 中置記法計算機專案

- 中置記法計算機請參閱上一章。本小節只是將掃描函式 yylex() 從文法檔案 calc.y 抽出，另外製作一個 calc.l 規格檔後再將它們結合起來而已。
- 本專案存於 ***C:\plone\incalc*** 目錄。
- 在 flex 的規格檔 calc.l 中：
- ***#include "calc.tab.h"***
- 這個表頭檔由 Bison 所建立，您必須將它引入 calc.l 檔案裡頭，否則執行 flex.exe 時會產生錯誤的。

15.4 符號計算機專案

- 符號計算機請參閱上一章。本小節只是將掃描函式 yylex() 從文法檔案 calc.y 抽出，另外製作一個 calc.l 規格檔後再將它們結合起來而已。
- 本專案存於 ***C:\plone\symcalc*** 目錄。
- 在 flex 的規格檔 calc.l 中：
- ***#include "calc.tab.h"***
- 這個表頭檔由 Bison 所建立，您必須將它引入 calc.l 檔案裡頭，否則執行 flex.exe 時會產生錯誤的。

15.5 函式計算機專案

- 函式計算機請參閱上一章。本小節只是將掃描函式 yylex() 從文法檔案 calc.y 抽出，另外製作一個 calc.l 規格檔後再將它們結合起來而已。
- 本專案存於 ***C:\plone\mfcalc*** 目錄。
- 在 flex 的規格檔 calc.l 中：
- ***#include "calc.tab.h"***
- 這個表頭檔由 Bison 所建立，您必須將它引入 calc.l 檔案裡頭，否則執行 flex.exe 時會產生錯誤的。

15.6 計算機編譯器專案

- 計算機編譯器專案的引入 cal.h 表頭檔是相關程式都會用到的共用資料，包括常數值、變數、運算子等資料。
- ***enum enumTag { enumNUM, enumVAR, enumOPR };***
- 這是定義一個列舉型態「enum enumTag」，含三個元素分別代表常數值 enumNUM、變數 enumVAR、運算子 enumOPR 等。
- ***struct numTag { int num; };***
- 這是常數值的結構，只含一個 num 整數型態成員（member）。
- ***struct varTag { int var; };***
- 這是變數的結構，只含一個 var 整數型態成員。

```
struct oprTag
{
  int oper;
  struct nodeTag *op[2];
};
```

- 這是運算子的結構，含二個成員，整數 oper 表運算子，op[ ] 節點指標，指向所要操作的運算元，這種運算元指標共有兩個，分別為 op[0] 、op[1]。

```
struct nodeTag
{
  enum enumTag type;
  union
  {
    struct numTag num;
    struct varTag var;
    struct oprTag opr;
  };
};
```

- 這是運算元的結構，一個列舉型態的成員 type。一個 union 成員。
- 若 type 值為 enumNUM 表示常數運算元，其值為 num。
- 若 type 值為 enumVAR 表示變數運算元，其值為 var。
- 若 type 值為 enumOPR 表示運算子運算元，其值為 opr。

語彙分析 flex 規格檔 cal.l 說明如下

- 正規運算式 [a-z] 表示變數，為了單純化起見本專案規定變數只限用 a 至 z 間的一個字母而已。我們在 Bison 文法檔案 cal.y 裡對於符號的語意值型態作如下的宣告：
- ***%union***
- ***{***
- ***int numValue;***
- ***char varIndex;***
- ***struct nodeTag *nodePtr;***
- ***};***
- ***%token <numValue> NUM***
- ***%token <varIndex> VAR***
- 表示符記 NUM 的型態為 numValue 成員的整數型態，符記 VAR 的型態為成員 varIndex 的整數型態。

- [a-z] {
- yylval.varIndex = *yytext - 'a';
- return VAR;
- }
- 因此語意值整體變數 yylval 是一個 union 同位結構，它包含三個成員 numValue、varIndex、nodePtr，這三個成員都佔同一塊記憶體。當辨認出一個小寫的英文字母時，將該字母「*yytext」減去 'a' ASCII 字元值（97）後存入 yylval 結構裡的 varIndex 成員，因此 'a'-'z' 其值的範圍為 0 至 25。

- ***[0-9]+ {***
- ***yylval.numValue = atoi(yytext);***
- ***return NUM;***
- ***}***
- 當辨認出 NUM 常數時，將字串 yytext 透過 C 語言的 atoi() 函式轉換成浮點數後存入 yylval 的 numValue 成員。
- ***[-+()=/*;] { return *yytext; }***
- 這是單字元的符記，只單純的傳回其 ASCII 字元值。
- ***"print" return PRINT;***
- 這是 print 指令，用於列印變數值，傳回 PRINT 符記。

- Bison 文法檔 cal.y 可說是本專案的重心，您應該還記得 Bison 的文法檔包括四個段落：前導段落、Bison宣告段落、文法規則段落、結尾段落。
- 前導段落包含在「%{」與「%}」之間。標明引入檔及 C 語言的一些宣告，包括函式原型（prototype）宣告以及變數宣告。
- Bison宣告段落以「%」開頭，一直到「%%」為止。

```
%union
{
  int  numValue;
  char varIndex;
  struct nodeTag *nodePtr;
};
%token <numValue> NUM
%token <varIndex> VAR
%token PRINT
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%type <nodePtr> stmt expr stmt_list
```

- 這是 Bison宣告，表示符號的語意值型態有三種。符記 NUM 為整數，符記 VAR 為整數，PRINT 符記沒有宣告型態，表示它不需語意值。
- 非終端符號的 stmt、expr、stmt_list 均為結構指標型態。
- UMINUS 無結合，'+'、'-'、'*'、'/' 均屬左結合，UMINUS 最優先、其次為 '*' 與 '/'，再其次為 '+' 與 '-'。

- 兩個「%%」之間為文法規則段落。這些文法規則用到了定義在結尾段落的幾個函式，num()、var()、opr() 分別處理常數、變數、運算子。
- expr: NUM { $$ = num($1); }
- 上述文法規則將 $1 第一個組件（component）NUM 傳給 num() 函式的 val 參數，num() 函式的程式碼如下：
- ```
  struct nodeTag *num(int val)
  {
    struct nodeTag *p=malloc(
                  sizeof(struct nodeTag));
    p->type = enumNUM;
    p->num.num = val;
    return p;
  }
  ```
  ```

- 首先宣告一個「struct nodeTag *」指標變數 p，並取得一塊結構大小的記憶體給 p 指標變數。
- 1. 將 NUM 的列舉型態 enumNUM 值存入 p->type。
- 2. 將 NUM 值存入 p->num.num。
- 3. 最後將此指標 p 傳回給非終端符號 expr。
- 這個傳回的 p 指標變數指定給文法左邊的 expr 非終端符號，以符號 $$ 表示。
- expr: VAR { $$ = var($1); }
- 上述文法規則將 $1 第一個組件 VAR 傳給 var() 函式的 val 參數，var() 函式的程式碼如下：

```
struct nodeTag *var(int val)
{
  struct nodeTag *p=malloc(sizeof(struct nodeTag));
  p->type = enumVAR;
  p->var.var = val;
  return p;
}
```

- 首先宣告一個「struct nodeTag *」指標變數 p，並取得一塊結構大小的記憶體給 p 指標變數。
- 1. 將 VAR 的列舉型態 enumVAR 值存入 p->type。
- 2. 將 VAR 值存入 p->var.var。
- 3. 最後將此指標 p 傳回給非終端符號 expr。
- 這個傳回的 p 指標變數指定給文法左邊的 expr 非終端符號，以符號 $$ 表示。

- expr: expr '+' expr { $$ = opr('+', $1, $3); }
-
- 上述文法規則將 $2 第二個組件 ‘+’、第一個組件 expr（$1）、第三個組件 expr（$3），傳給 opr() 函式的三個參數 oper、opn1、opn2，opr() 函式的程式碼如下。
- 因為運算元個數可能為 1（UMINUS、PRINT），也可能為 2（'+'、'-'、'*'、'/'），是一個變動的值，因此所必需取得的記憶體大小也就不同，為了單純起見我們固定提供兩個運算元的空間，若只有一個運算元則第二個不用而已。

```
struct nodeTag *opr(int oper, struct nodeTag *opn1,
                    struct nodeTag *opn2)
{
  struct nodeTag *p=malloc(sizeof(struct nodeTag));
  p->type = enumOPR;
  p->opr.oper = oper;
  p->opr.op[0] = opn1;
  p->opr.op[1] = opn2;
  return p;
}
```

- 首先宣告一個「struct nodeTag *」指標變數 p，並取得一塊結構大小的記憶體給 p 指標變數。
- ***1. 將 OPR 的列舉型態 enumOPR 值存入 p->type。***
- ***2. 將 oper 運算子的值存入 p->opr.opr。***
- ***3. 將左運算元指標 opn1 存入 p->opr.op[0]。***
- ***4. 將右運算元指標 opn2 存入 p->opr.op[1]。***
- ***5. 最後將此指標 p 傳回給非終端符號 expr。***
- 這個傳回的 p 指標變數指定給文法左邊的 expr 非終端符號，以符號 $$ 表示。

- function: function stmt { ex($2); }
- 在 cal.y 的 Bison 文法檔案裡的文法規則中，當敘述 stmt 非終端符號結束時，我們想要解釋此一個敘述，可呼叫 ex() 函式，將 stmt 指標傳給它，當它的參數，其程式碼架構如下。

```
int ex(struct nodeTag *p)
{
  if  (p==NULL)  return 0;
  switch(p->type)
  {
    case enumNUM: return  p->num.num;
    case enumVAR: return  sym[p->var.var];
    case enumOPR:
    /*略*/
  }
  return  0;
}
```

- 當 p 節點指標為 NULL 值時（不指到任何節點）傳回 0 值。**若為常數則傳回常數**值，若為變數則傳回變數值。傳回常數值容易了解，傳回變數值可能比較不容易了解，您知道變數值存在那裡嗎？我們先來執行一個簡單的測試檔案 test 如下：
- C:\plone\jaycalc> cali test 1 <Enter>
- file "test" contents :
- f=5+4;
- print f;
- after yyparse(), print as followings:
- num() enumNUM 5
- num() enumNUM 4
- opr() enumOPR '+'
- var() enumVAR 'f'
- opr() enumOPR '='
- var() enumVAR 'f'
- opr() enumOPR 260
- 9
- 測試檔 test 只含兩列：「f=5+4;」及「print f;」。

- 命令列「cali test 1」第三個引數為 1，表示要除錯（debug=1）。

- 第一列「f=5+4;」被轉換成「5、4、+、f、=」，其和存入 f 變數處。您還記得我們的變數規定只能使用 a 至 z 間的一個字母而已，其索引值範圍為 0 至 25，a 相當於索引值 0 位置，z 相當於索引值 25 位置，其他字母在 0-25 中間，因此設計一個含有 26 個元素的整數陣列sym[26]，用來儲存這 26 個變數的值。

- 從執行的結果可以看出 degree 檔裡的三個敘述產生三棵樹。第一個敘述「c=30;」產生的樹如圖 15.2 所示。第二個敘述「f=c*9/5+32;」產生的樹如圖 15.3 所示。第三個敘述「print f;」產生的樹如圖 15.4 所示。