第十四章 Bison 使用

- Bison 是 GNU（Gnu's Not Unix）Free Software Foundation（自由軟體基金會）的一個計劃，希望能夠取代 UNIX 中 yacc 的公用程式。
- GNU 這個組織的目標就是要創造一個類似 UNIX 的作業系統，並提供原始碼讓人們自由取得。雖然 GNU 不屬公用領域（public domain），但它所提供的軟體任何人都可自由取用，因為它的軟體授權書（license）就是要求所有人都要可以自由取得該項軟體。因此任何人都可透過下列網站取得 Bison 軟體。
- ***http://www.gnu.org/software/bison/***
- Bison 是跟據早期的 Berkeley 版本改寫的軟體，它與 yacc 原則上是相容的，不過細節上總有小部份不同，本章介紹 Bison 的操作。

14.1 Bison語法要點

- Bison 從輸入資料取得上下文無關文法（context-free grammar）的規格資料，產生一個 C 語言的原始程式，用於辨認 LALR(1) 語法，它是下列文字的縮寫：
- ***Look-Ahead Left-to-right-parse Rightmost-derivation***
- 預看式左往右剖析最右優先導出。
- Bison 文法規格檔（grammar file）包含四個段落，格式如下。
- ***%{***
- ***前導段落（Prologue section）***
- ***%}***
- ***Bison宣告段落（Bison declarations section）***
- ***%%***
- ***文法規則段落（Grammar rules section）***
- ***%%***
- ***結尾段落（Epilogue section）***

- C 語言的註解「/*...*/」可以散佈在任何段落。GNU 所延伸的註解「//」其效力只到該列列尾。
- 「前導段落」包含巨集（macro）定義、函式（function）宣告、以及使用於「文法規則段落」裡的變數（variable）宣告。
- 在前導段落裡的定義及宣告會拷貝至 Bison 所產生 C 語言剖析原始程式 yyparse() 函式的前端。您可使用 #include 指引將 C 語言的表頭檔（header file）引入您的程式裡，若不需要任何的 C 語言宣告，可省略「%{..%}」這個段落。
- 前導段落間雜著Bison宣告段落，可出現多次，讓彼此之間可以互相參考。

Bison宣告段落

- 「%union」屬於Bison宣告段落，所使用的「struct nodeTag *」型態宣告在前導段落裡的 calc.h 表頭檔裡，前導段落裡的 yylex() 函式會用到屬於Bison宣告段落裡的 %union 宣告。

```
%{
  #include <stdio.h>
  #include <ctype.h>
  #include "calc.h"
%}
%union {
    double val;
    struct nodeTag *nodeptr;      /*宣告於calc.h表頭檔*/
  }
%{
    int yylex(void);                    /*宣告函式原型*/
    void yyerror(char const *);         /*宣告函式原型*/
%}
```

Bison 文法規格檔

- ***「Bison宣告段落」*** 包含宣告，用於定義終端符號（terminal）、非終端符號（nonterminal）、標明優先順序（precedence）等等。
- ***「文法規則段落」*** 包含一個或多個 Bison 語法規則，就只有語法規則，沒有其他的了。
- ***「結尾段落」*** 會拷貝至 Bison 所產生 C 語言剖析原始程式函式的後端。在這個段落您可擺上您想要的函式，例如 yylex() 及 yyerror() 函式定義經常會擺在這個段落，不過 C 語言要求在使用函式之前必須要宣告，我們的剖析函式 yyparse() 已經在前面使用 yylex() 函式，而您在 yyparse() 函式後面才來定義，會產生語法錯誤的，因此您有必要在前導段落裡先宣告此二函式的原型（prototype），如上例。

14.2 符號

- Bison 文法裡的符號代表某種語言的文法種類（classification）。
- ***終端符號***，也是眾所週知的符記型態，代表相當的語法種類。您在文法規則段落裡所使用的終端符號表示該符記是被允許的。在 Bison 所產生的 C 語言剖析原始程式裡頭符號以整數表示，因此 yylex() 函式傳回一個整數表示某一個符記已經從輸入媒體讀入，您並不須知道那一個符記整數編號到底是多少，您只須曉得該終端符號代表該符記就行了。
- 一個 ***非終端符號*** 代表一個語法上的種類，相當於一個語法群組。非終端符號的名稱可用於撰寫語法規則，習慣上使用小寫字母的名稱。名稱由字母、阿拉伯數字、底線及句點組成，阿拉伯數字不能當第一個名稱的字元，句點只限用於非終端符號名稱。

撰寫語法規則有下列三種方式

- ***1.*** 符記名稱使用大寫字母，依 C 語言識別字規則命名。名稱必須以 %token 宣告。如下例：
- %token NUM ID
- ***2.*** 字元符記以 C 語言字元常數表示。以逸出順序表示也可以。例如 '+' 字元表示 '+' 符記，'\n' 表示新列。
- ***3.*** 字串符記以 C 語言字串常數表示。不適用於 yacc。例如 "<=" 字串表示 "<=" 符記。

yylex() 函式

- 語彙分析 yylex() 函式傳回的是終端符號其中的一個，但當讀至輸入檔檔尾時傳回 0 值或負值。您用什麼方式在文法規則裡撰寫符記，就用該方式在 yylex() 函式裡撰寫符記。字元符記其值就是該字元本身的值，例如字元符記 '+' 其值就是 43 或十六進位值 0x2B，yylex() 函式就可用該值去處理。
- 若 yylex() 定義於另一個檔案，您必須提供一個符記名稱及其代表的整數編號對照表，如此 yylex() 函式才能工作。
- 這時您要使用 Bison 的「-d」選項，就可產生一個相對應的對照表，如下例：
- C:\plone\ch14> bison -d symcalc.y <Enter>
- 會產生一個 symcalc.tab.h 表頭檔，含 yylex() 函式的檔案，只須將此表頭檔引入該 yylex() 函式就可以了。

14.3 文法規則的語法

- Bison 的文法規則，其語法如下：
- ***result: components ... ;***
- 這裡的 result 是某一個非終端符號，components 表示組件。「...」表示重複。「;」表示該文法規則結束。組件可為終端符號或非終端符號。如下例：
- ***expr : expr '+' expr ;***
- 表示右邊兩個 expr 型態的組件以 '+' 符記相結合而成一個較大的非終端符號 expr。

同一個 result 可以表示多個文法規則

- 同一個 result 可以表示多個文法規則，文法規則間以垂直棒「|」隔開，格式如下：
- ***result***
- ***: rule1-components ...***
- **| *rule2-components ...***
- ***...***
- ***;***
- 若文法規則裡的組件是空白，表示 result 非終端符號匹配空字串，如下例：
- ***input***
- ***: /*empty*/***
- **| *input line***
- ***;***
- 表示 input 非終端符號可以匹配空字串，或多個 line，空字串習慣上以 /*empty*/ 註解表示。

14.4 遞迴規則

- 一個文法規則稱為遞迴規則，表示 result 非終端符號同時出現在文法規則的右手方。如下例：
- input
- : /*empty*/
- | input line
- ;
- 非終端符號 input 同時出現在文法規的左右方，input 可能擴展成如下的各種情形。
- /*empty*/
- /*empty*/ line
- /*empty*/ line line
- /*empty*/ line line line
- ...
- /*empty*/ line line line ... line
- 因為 input 出現在 line 左邊，故稱為 ***左遞迴***。

14.5 定義語言語意

- 一個語言的文法規則只決定它的語法（syntax）而已。一個語言的語意（semantic）決定於連繫在各個符記及群組的語意值，以及當各個群組被辨認時所採取的行動。
- 對於一個簡單的程式只須用到一個語意值的型態就可以了，Bison 內定的語意值型態為 int 整數型態。但您可以透過下列的定義改變語意值型態為指定的型態。
- ***#define YYSTYPE 指定的型態***
- 如下例改變為倍精確浮點數 double 型態。
- ***#define YYSTYPE double***
- 這個定義必須擺在 Bison 文法檔案的前導段落。

對於不同的符記或群組給不同的型態

- 但對於大多數的程式來說，您有需要對於不同的符記或群組給不同的型態。Bison 提供下列兩種方式：
- ***1. 透過 %union 宣告標明各種可能的資料型態。***
- ***2. 透過 %token 標明各種符記（終端符號）的資料型態。***
- ***透過 %type 標明各種群組（非終端符號）的資料型態。***
- 伴隨者文法規則的動作包含 C 語言的程式碼，當該群組被辨認出來時所要執行的程式碼。大部份的動作是計算該群組的語意值。其程式碼必須以大括號括起來。動作可以擺在文法規則的任何位置，大部份文法規則只在規則的尾端擺放一個動作，如下例。

文法規則

- expr
- : NUM { $$ = $1; }
- | expr '+' expr { $$ = $1 + $3; }
- ;
- **第一條文法規則** 說明，當辨認出 NUM 符記時，將 NUM 的語意值給予左邊的 expr 文法群組。
- **第二條文法規則** 說明，當辨認出 expr '+' expr 群組時，將第一個組件 expr 語意值加上第三個組件 expr 語意值的和給予左邊的非終端符號 expr。
- **「$n」表示右邊第 n 個組件**（component），「$$」表示左邊的非終端符號。這些語意值的運算假設它們都屬於同一個資料型態，若它們間的資料型態不同，那就必須清楚的表示出來。

```
%union {
  double val;
  struct nodeTag *nodeptr;
  }
%token <val>     NUM
%token <nodeptr> VAR
%type  <val>     expr
/*略*/
expr
  : NUM               { $$ = $1; }
  | VAR               { $$ = $1->value.var; }
  | VAR '=' expr    { $$ = $3; $1->value.var = $3; }
  | expr '+' expr     { $$ = $1 + $3; }
  ;
```

- 宣告 NUM 及 expr 均屬於 %union 宣告裡的「double」型態。宣告 VAR 屬於 %union 宣告裡的「struct nodeTag *」型態。

14.6 定位追蹤

```
expr
 : NUM { $$ = $1; }
 | expr '/' expr
   {
     if ($3)
       $$ = $1 / $3;
     else
     {
       $$ = 1;
       fprintf (stderr,
         "%d.%d-%d.%d: division by zero ",
         @3.first_line, @3.first_column,
         @3.last_line, @3.last_column);
     }
   }
 ;
```

14.7 Bison 宣告

- Bison 文法檔案裡的「Bison宣告段落」定義使用於文法規則裡的符號，以及語意值的資料型態。
- 所有符記名稱都必須宣告型態（單字元常數符記例外）。需要語意值的非終端符號都必須宣告其型態。
- 符記名稱宣告型態，其格式如下：
- ***%token name***
- Bison 會透過 #define 指引將它轉換成一個 name 名稱的巨集，以便 yylex() 函式使用。

```
%union {
 double val;
 struct nodeTag *nodeptr;
 }
%token <val>     NUM
%token <nodeptr> VAR
%type  <val>     expr
%right '='
%left '-' '+'
%left '*' '/'
%left NEG
%right '^'
```

- 符記 NUM 以及群組（非終端符號）均宣告為 double 資料型態，但符記 VAR 卻宣告與 double 不同的 struct nodeTag * 指標型態。
- 請注意它們都以角括號括起來。%right 宣告右結合（association），宣告 '=' 及 '^' 運算子均為右結合。%left 宣告為左結合，'-'、'+' 、'*'、'/'、NEG 都屬於左結合。宣告在愈底下的其優先順序愈高，乘除宣告在加減之下，表示先乘除後加減。
- 宣告運算子的優先順序可使用 %left、%right、%nonassoc 等宣告。其語法如下：
- ***%left 符號 ...***
- 或則：
- ***%left <type> 符號 ...***
- 宣告群組（非終端符號）的資料型態其格式如下：
- ***%type <type> 非終端符號 ...***
- 宣告開始剖析的符號，其格式如下：
- ***%start 符號***

14.8 剖析函式 yyparse()

- 您要呼叫 yyparse() 函式執行剖析的工作。這個函式讀取符記、辨認符記並執行相對應的動作（C 語言程式碼），當讀取符記結束時、或剖析時產生錯誤，這時才傳回適當的值，返回呼叫者程式裡頭。
- yyparse() 函式的簽名格式如下：
- ***int yyparse(void)***
- 正常結束時傳回 0 值，失敗時傳回 1 值，記憶體不足時傳回 2 值。使用 YYACCEPT 巨集導至剖析函式立即傳回 0 值，使用 YYABORT 巨集導至剖析函式立即傳回 1 值。

14.9 語彙分析函式 yylex()

- 語彙分析函式 yylex() 從輸入資料流辨認符記並傳回給剖析函式。Bison 並不會自動產生此函式，您必須自己準備妥當。
- 在一個簡單的程式裡頭，yylex() 函式通常擺在 Bison 文法規則的後面。若 yylex() 定義於另一個檔案，您必須提供符記與整數編號對照表給語彙分析函式，還好當您選用 Bison 的 -d 選項時它會自動產生一個對照表，您只須將該表引入含有 yylex() 函式的檔案就可以了。
- yylex() 函式會傳回一個正整數值，對應所讀取的符記，當讀至輸入檔檔尾時傳回一個 0 或負整數值。

- 當文法規則裡的符記屬於名稱時，該名稱在剖析函式裡頭變成一個巨集定義，其定義為該符記的整數編號，因此 yylex() 函式可以使用該巨集名稱表示該整數編號。
- 當文法規則裡的符記屬於字元常數時，該字元常數的值代表該符記。

14.10 符記的語意值

- 符記的語意值必須儲存在整體變數 yylval 裡頭，當您只使用一種資料型態時，整體變數 yylval 就是那一個型態。因此若內定為整數型態，您在 yylex() 函式裡撰寫的程式碼如下：
- …
- ***yylval = value;　/*將value疊入Bison堆疊*/***
- ***return NUM;　/*傳回符記型態（整數編號）*/***

- %union {
- double val;
- struct nodeTag *nodeptr;
- }
- ...
- scanf("%lf", &yylval.val);
- return NUM;
- ...
- yylval.nodeptr = s;
- return s->sym;
- 上例宣告兩種型態，浮點數及指標型態。
- NUM 屬浮點數，因此符記的語意值為 yylval.val。
- 識別字 s->sym 屬指標，符記的語意值為 ylval.nodeptr。
- 各屬不同的型態。

14.11 Bison宣告彙總

- %union
- 宣告所有語意值的資料型態。
- %token
- 宣告一個沒有標明優先順序及結合的終端符號。
- %right
- 宣告一個屬於右結合的終端符號。
- %left
- 宣告一個屬於左結合的終端符號。
- %nonassoc
- 宣告一個不屬任何結合的終端符號。
- %type
- 宣告一個語意值的非終端符號資料型態。
- %start
- 標明剖析的開始符號。
- %expect
- 標明「移位/簡化（shift/reduce）」衝突（conflict）個數。

Bison 指引

- %debug
- 此指引相當於宣告 YYDEBUG 的值為 1。
- #define YYDEBUG 1
- 並將此巨集宣告擺在前導段落。
- 您也可以在編譯時使用 -DYYDEBUG=1 選項。
- 或則使用 -t 或 --debug 選項。均可達到除錯的目的。
- %defines
- 輸出一個含有符記名稱的巨集定義及一些其他宣告的 C 語
- 言表頭檔。若輸出的剖析器檔名為 "name.c" 則本表頭檔名
- 為 "name.h"。
- %destructor
- 標明 Bison 如何重新使用被拋棄符號的記憶體。

- %file-prefix="prefix"
- 標明輸出檔案字首的名稱。
- 此指引相當於剖析檔 "prefix.y" 所產生的輸出檔
- 為 "prefix.tab.c"。
- %locations
- 產生處理符號位置的程式碼。
- %name-prefix="prefix"
- 以 prefix 取代 yy 字首。
- %no-parser
- 在剖析檔裡不含任何的 C 語言程式碼，只建立資料表而
- 已。在剖析檔裡只包含 #define 指引以及靜態變數宣告。
- 輸出一個文法規則段落動作程式碼檔，檔名為 "file.act"。
- %no-lines
- 在剖析檔裡不要建立 #line 資料。

- %output="file"
- 標明剖析檔名為 "file"。
- %pure-parser
- 需要一個單純（可重新進入 reentrant）的剖析程式。
- %token-table
- 在剖析檔裡建立一個符記名稱的陣列，陣列名稱為
- yytname。前三個元素為內定的 “$end”、“error”、“$undefined”。
- Bison 同時建立下列的巨集供您使用：
- YYNTOKENS 最大的符記編號在加一（從0算起）。
- YYNNTS 非終端符號個數。
- YYNRULES 文法規則個數。
- YYNSTATES 剖析狀態個數。
- %verbose
- 輸出一個額外的檔案，包含剖析狀態時一些冗長的描述，以及
- 在該狀態時對於往前看到的符記所作的事情。
- %yacc
- 相當於 --yacc 選項，模擬 Yacc。

14.12 執行 Bison

- 使用下列格式在作業系統的命令列執行 Bison：
- ***bison infile <Enter>***
- infile 為 Bison 文法檔名，習慣上使用「.y」副檔名。所產生的剖析檔名會將「.y」改變為「.tab.c」並移出前導的目錄名稱。
- 例如「calc.y」文法檔名，其相對應的剖析檔名為「calc.tab.c」。「C:\plone\ch14\calc.y」文法檔名，其相對應的剖析檔名也是「calc.tab.c」。

- ***執行 bison 時可使用下列的選項：***

- '-h'
- '--help'
- 列印所有選項。
- 'V'
- '--version'
- 列印版本編號後結束執行。
- '--print-localedir'
- 列印目錄名稱。
- '-y'
- '--yacc'
- 相當於「-o y.tab.c」， 輸出剖析檔名為 "y.tab.c",
- 其他檔名為 "y.output" 及 "y.tab.h"。
- '-S file'
- '--skeleton=file'
- 標明使用結構的檔名。

- '-t'
- '--debug'
- 在剖析檔裡將 YYDEBUG 巨集定義為 1。
- '--locations'
- 相當於 %locations 宣告。
- '-p prefix'
- '--name-prefix=prefix'
- 相當於 %name-prefix="prefix" 宣告。
- '-l'
- '--no-lines'
- 相當於 %no-lines 宣告。
- 'n'
- '--no-parser'
- 相當於 %no-parser 宣告。
- '-k'
- '--token-table'
- 相當於 %token-table 宣告。

- '-d'
- '--defines'
- 相當於 %defines 宣告。
- '--defines=defines-file'
- 相當於 %defines 宣告，但存入指定的 defines-file 檔案。
- '-b file=prefix'
- '--file-prefix=prefix'
- 相當於 %verbose 宣告。
- '-r things'
- '--report=things'
- 輸出一個額外檔案包含以逗號隔開項目的冗長描述，項目如下。
- state
- 描述文法、衝突、LALR 自動機。
- look-ahead
- 隱含state及每一個文法規則往前看符記集合引數描述。
- itemset
- 隱含state及look-ahead項目。

- '-v'
- '--verbose'
- 相當於 %verbose 宣告。
- '-o file'
- '--output=file'
- 指定剖析檔名為 file。
- 其他檔名依 '-v' 及 '-d' 選項而定。
- '-g'
- 輸出一個 Bison 所使用 LALR(1) 文法的 VCG 定
- 義檔案。若文法檔名為 “calc.y” 則其 VCG 檔名
- 為 "calc.vcg"。
- '--grapg=graph-file'
- 與 '-g' 同，不過輸出至指定的 graph-file 檔案。

14.13 後置記法計算機

- 前面說明許多 Bison 的語法，您一定會覺得枯燥乏味了。舉幾個例子來說明 Bison 的用法。
- 運算子（operator）擺在運算元（operand）的後面，稱為後置記法，也稱為反波蘭記法（reverse polish notation），如下例：
- ***12 13 +***
- 運算元為 12 與 13，運算子為 '+'，計算結果為 27。
- 又如下例：
- ***33 44 + 5 6 7 *+-***
- 33+44 得 77，6*7 得 42，5+42 得 47，77-47 得 30，結果為 30。其執行順序如下：
- ***((33 44 +) (5 (6 7 *) +) -)***

Bison 文法檔案 rpncalc.y 分析如下

- 其 Bison 文法檔案 rpncalc.y 分析如下。
- ***/********* 後置記法計算機 (rpncalc.y) *********/***
- 這是 C 語言的註解，可擺在任何位置。
- %{
- #include <stdio.h> /*C**語言標準輸入輸出表頭檔***/
- #include <ctype.h>
- #include <math.h>
- #define YYSTYPE double /*巨集定義*/
- int yylex(void); /*函式原型*/
- void yyerror(char const *); /*函式原型*/
- %}
- 這是「前導段落」，包含巨集定義及函式宣告。巨集 YYSTYPE 定義

```
input
  : /*empty*/
  | input line
  ;
line
  : '\n'
  | expr '\n'    { printf ("\t%.10g\n", $1); }
  ;
expr
  : NUM          { $$ = $1; }
  | expr expr '+' { $$ = $1 + $2; }
  | expr expr '-' { $$ = $1 - $2; }
  | expr expr '*' { $$ = $1 * $2; }
  | expr expr '/' { $$ = $1 / $2; }
  | expr expr '^' { $$ = pow ($1, $2); }
  | expr 'n'     { $$ = -$1; }
;
```

- 這是「文法規則段落」，包含三個非終端符號input、line、expr等的文法規則。包含在大括號裡的是相對應的動作。「$$」表示左手邊的非終端符號，「$n」表示右手邊第 n 個組件（component）。

- ***expr : expr expr '+' { $$ = $1 + $2; }***
- 表示第一個組件 expr 的語意值加上第二個組件 expr 的語意值後存入左手邊非終端符號 expr，並當它的語意值。'+' 是第三個組件 $3。
- ***void yyerror(char const *s) /*錯誤時呼叫此函式*/***
- ***...***
- ***int yylex (void)***
- ***...***
- ***int main (int argc, char *argv[])***
- ***{***
- ***printf("輸入資料檔\"rpncalc.txt\"內容如下：\n");***
- ***system("TYPE rpncalc.txt");***
- ***printf("透過bison的yyparse()逐一剖析如下：\n");***
- ***yyparse();***
- ***return 0;***
- ***}***
- 這是「結尾段落」，直接拷貝至 Bison 所產生的剖析程式後端。

- 請注意在 yylex() 函式裡頭下列敘述：
- ***scanf("%lf", &yylval); /*再從輸入資料流以浮點數讀回*/***
- ***return NUM; /*傳回NUM整數編號*/***
- 它是已經辨認為 NUM 符記時，從輸入資料流以倍精確浮點數格式讀入至語意值整體變數 yylval 裡頭，因為「#define YYSTYPE double」已經定義巨集 YYSTYPE 為 double 型態，因此 NUM 符記的語意值屬於 double，因此以 %lf 格式讀入。
- main() 主函式為程式開始執行的函式，透過作業系統的 TYPE 命令將指定的 Bison 文法檔案 rpncalc.txt 內容輸出至螢幕，然後呼叫剖析函式 yyparse() 開始剖析。剖析過程當中會執行動作中的程式碼而輸出非終端符號的 $$ 語意值。

- 為了方便執行，建立一個 rpnrun.bat 批次檔，只要在命令列輸入批次檔名就可執行。
- ***C:\plone\ch14> rpnrun.bat <Enter>***
- 批次檔 rpnrun.bat 內容如下：
- ***bison rpncalc.y***
- ***gcc rpncalc.tab.c -o rpncalc.exe***
- ***rpncalc.exe < rpncalc.txt***
- 首先透過 bison 將輸入的文法檔案 rpncalc.y 轉換成原始程式，副檔名內定為「tab.c」，整個檔名為 rpncalc.tab.c。接著使用 gcc 編譯器將它編譯成 rpncalc.exe 可執行檔案。最後執行該程式，將內定從鍵盤輸入的轉而從本文檔 rpncalc.txt 輸入。

- **我們來看看 Bison 根據我們在「文法規則段落」所提供的文法如何剖析。以句點符號「.」左邊表示堆疊，最靠近句點符號的為堆疊頂端，最遠處為底端。在句點符號右邊的為輸入資料流，即 rpncalc.txt 檔案，最靠近句點符號的為目前所要讀取的字元，其剖析步驟如下。**
-
- **表 14.1 後置記法計算機剖析步驟**
-
- **步驟 堆疊.輸入資料流** 說明
- ---- ---------- ----------
- 1 . 12 13 + ₩n 33 44 + 5 6 7 *+-₩n EOF **輸入資料流**
- 2 12 . 13 + ₩n 33 44 + 5 6 7 *+-₩n EOF **移位**shift
- 3 NUM . 13 + ₩n 33 44 + 5 6 7 *+-₩n EOF **簡化**reduce
- 4 expr . 13 + ₩n 33 44 + 5 6 7 *+-₩n EOF **簡化**
- 5 expr 13 . + ₩n 33 44 + 5 6 7 *+-₩n EOF **移位**
- 6 expr NUM . + ₩n 33 44 + 5 6 7 *+-₩n EOF **簡化**
- 7 expr expr . + ₩n 33 44 + 5 6 7 *+-₩n EOF **簡化**
- 8 expr expr + . ₩n 33 44 + 5 6 7 *+-₩n EOF **移位**
- 9 expr . ₩n 33 44 + 5 6 7 *+-₩n EOF **簡化**
- 10 expr ₩n . 33 44 + 5 6 7 *+-₩n EOF **移位**

- 11 line . 33 44 + 5 6 7 *+-₩n EOF 簡化
- 12 line 33 . 44 + 5 6 7 *+-₩n EOF 移位
- 13 line NUM . 44 + 5 6 7 *+-₩n EOF 簡化
- 14 line expr . 44 + 5 6 7 *+-₩n EOF 簡化
- 15 line expr 44 . + 5 6 7 *+-₩n EOF 移位
- 16 line expr NUM . + 5 6 7 *+-₩n EOF 簡化
- 17 line expr expr . + 5 6 7 *+-₩n EOF 簡化
- 18 line expr expr + . 5 6 7 *+-₩n EOF 移位
- 19 line expr . 5 6 7 *+-₩n EOF 簡化
- 20 line expr 5 . 6 7 *+-₩n EOF 移位
- 21 line expr NUM . 6 7 *+-₩n EOF 簡化
- 22 line expr expr . 6 7 *+-₩n EOF 簡化
- 23 line expr expr 6 . 7 *+-₩n EOF 移位
- 24 line expr expr NUM . 7 *+-₩n EOF 簡化
- 24 line expr expr expr . 7 *+-₩n EOF 簡化

- 26 line expr expr expr 7 . *+-₩n EOF **移位**
- 27 line expr expr expr expr . *+-₩n EOF **簡化**
- 28 line expr expr expr expr * . +-₩n EOF **移位**
- 29 line expr expr expr . +-₩n EOF **簡化**
- 30 line expr expr expr + . -₩n EOF **移位**
- 31 line expr expr . -₩n EOF **簡化**
- 32 line expr expr - . ₩n EOF **移位**
- 33 line expr . ₩n EOF **簡化**
- 34 line expr ₩n . EOF **移位**
- 35 line line . EOF **簡化**
- 36 line line EOF . **移位**
- 37 input . **簡化**
- 38 **接受（剖析完成）**

14.14 中置記法計算機

- 運算子擺在兩個運算元之間，稱為中置記法（infix notation）。輸入的文法檔案 incalc.y 與上個例題後置記法 rpncalc.y 類似，所不同的只有文法規則與Bison宣告而已。
- %token NUM
- %left '-' '+'
- %left '*' '/'
- %left NEG /*negation--unary minus*/
- %right '^' /*exponentiation*/
- 這是「Bison宣告段落」，說明符記為 NUM，'-'、'+'、'*'、'/'、NEG 等均為左結合，'^' 為右結合。
- 其優先順序為 '^'、NEG、'*'、'/'、'-'、'+'，可見它滿足先乘除後加減的算術四則運算規則。

文法規則段落

```
input
: input line
| /*empty*/
;
line
: '\n'
| expr '\n'          { printf ("\t%.10g\n", $1); }
;
expr
: NUM                { $$ = $1; }
| expr '+' expr      { $$ = $1 + $3; }
| expr '-' expr      { $$ = $1 - $3; }
| expr '*' expr      { $$ = $1 * $3; }
| expr '/' expr      { $$ = $1 / $3; }
| '-' expr %prec NEG { $$ = -$2; }
| expr '^' expr      { $$ = pow ($1, $3); }
| '(' expr ')'       { $$ = $2; }
;
```

14.15 符號計算機

- 前面兩個例子所處理的都只有 NUM 數字符記而已，現在我們要加入變數符記 VAR。數字符記 NUM 的語意值 yylval 只是單純的 double 浮點數型態而已，變數符記 VAR 除了要儲存浮點數 double 型態的值之外，還要儲存其符記整數編號以及變數的名稱，因此只好使用 C 語言的結構型態來儲存這些相關的資料了。

```
struct nodeTag
{
  int  sym;                /*符記VAR整數編號*/
  char name[36];                /*符記VAR名稱*/
  union
  {
    double var;              /*符記VAR語意值*/
  } value;
  struct nodeTag *next;          /*下一個節點*/
};
```

- 請注意符記 VAR 的語意值必須擺在 union 同位結構裡，以搭配在文法檔案裡的 %union 宣告。對於每一個 VAR 符記提供一個節點，將所有的變數 VAR 節點存入一個連結串列結構的堆疊裡頭，堆疊頂點的指標命名為 nodestackTop，相關的四個函式其原型宣告如下：
- ***char *nodeToString(struct nodeTag *p);***
- ***char *nodestackToString();***
- ***struct nodeTag *putsym(char const *, int);***
- ***struct nodeTag *getsym(char const *);***

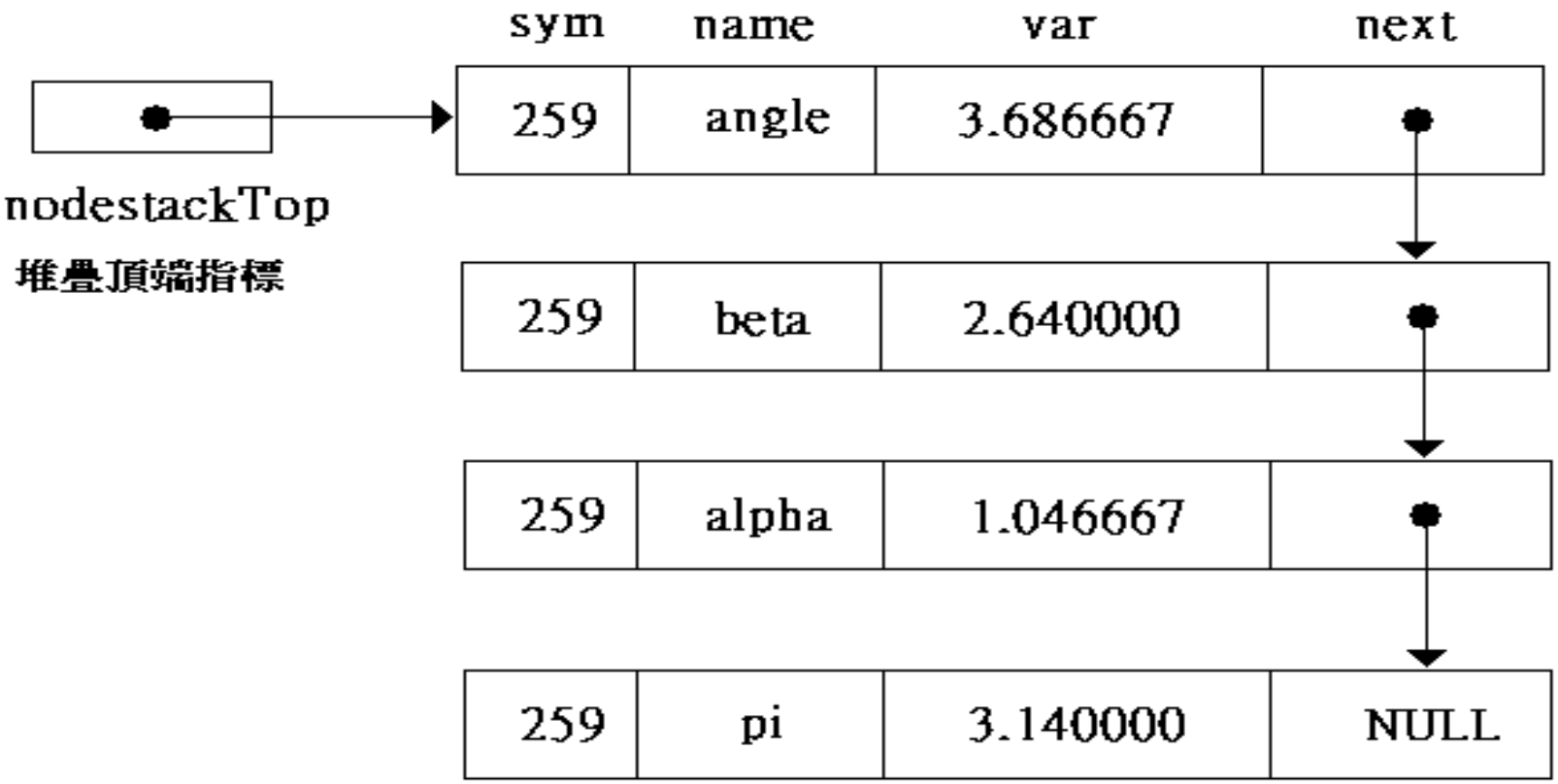
```
%union {
  double val;
  struct nodeTag *nodeptr;
  }
%token <val>     NUM
%token <nodeptr> VAR
%type  <val>     expr
%right '='
%left '-' '+'
%left '*' '/'
%left NEG
%right '^'
```

- 這是「Bison宣告段落」，%union 宣告語意值有兩種資料型態，一種為單純的浮點數 double 型態，另一種為「struct nodeTag *」結構指標型態。符記 NUM 及非終端符號 expr 屬於 <val> 型態（double），符記 VAR 屬於結構指標 <nodeptr> 型態（struct nodeTag *）。

- 【執行】
-
- C:\plone\ch14> symrun.bat <Enter>
- C:\plone\ch14>bison symcalc.y
- C:\plone\ch14>gcc symcalc.tab.c -o symcalc.exe
- C:\plone\ch14>symcalc.exe 0<symcalc.txt
- 輸入資料檔"symcalc.txt"內容如下：
- pi = 3.14
- pi
- alpha = pi*2/6
- alpha
- beta = (alpha+30/180)*3-1
- angle=alpha+beta

- 透過bison的yyparse()逐一剖析結果如下：
- putsym() p=0x003D2430 sym=259 name=pi var=0.000000
- 3.140000
- getsym() p=0x003D2430 sym=259 name=pi var=3.140000
- 3.140000
- putsym() p=0x003D2490 sym=259 name=alpha var=0.000000
- getsym() p=0x003D2430 sym=259 name=pi var=3.140000
- 1.046667
- getsym() p=0x003D2490 sym=259 name=alpha var=1.046667
- 1.046667
- putsym() p=0x003D24D0 sym=259 name=beta var=0.000000
- getsym() p=0x003D2490 sym=259 name=alpha var=1.046667
- 2.640000
- putsym() p=0x003D2510 sym=259 name=angle var=0.000000
- getsym() p=0x003D2490 sym=259 name=alpha var=1.046667
- getsym() p=0x003D24D0 sym=259 name=beta var=2.640000
- 3.686667
- ===== nodestackTop=0x003D2510 nodesize=56 =====
- p=0x003D2510 sym=259 name=angle var=3.686667 next=0x003D24D0
- p=0x003D24D0 sym=259 name=beta var=2.640000 next=0x003D2490
- p=0x003D2490 sym=259 name=alpha var=1.046667 next=0x003D2430
- p=0x003D2430 sym=259 name=pi var=3.140000 next=0x00000000

- 從執行的結果可以看出各變數的值如下：
- pi 3.140000
- alpha 1.046667
- beta 2.640000
- angle 3.686667
- 最後所構成的 VAR 堆疊如下圖所示。



14.16 函式計算機

- 我們希望計算機也可以執行 C 語言所提供的一些數學函式，例如求平方根的 sqrt() 函式，對數函式 log()，指數函式 exp()，三角正弦函式 sin()、餘弦函式 cos() 等等，也就是要將上一個符號計算機擴大功能的意思。
-
- 這些數學函式的名稱可以看成 VAR 的一種，事先將它們加入堆疊裡頭。

```
struct funTag
{
  char const *fname;
  double (*ftype) (double);
} funs[ ] = { "sin",  sin,
              "cos",  cos,
              "atan", atan,
              "ln",   log,
              "exp",  exp,
              "sqrt", sqrt,
            } ;
```

- 結構「struct funTag」包含函式名稱 fname 以及 (*ftype) 函式型態，一個浮點數 double 型態的參數，並傳回一個浮點數。funs[] 陣列變數提供六個數學函式。

- void stackfuns(void)
- {
- int i;
- struct nodeTag *p;
- for (i=0; i<sizeof(funs)/sizeof(struct funTag); i++)
- {
- p = putsym(funs[i].fname, FUN);
- p->value.funptr = funs[i].ftype;
- }
- }
- 函式 stackfuns() 將指定的六個數學函式透過 putsym() 函式逐一加入堆疊頂端。funs[i].fname 為第 i 個元素名稱，FUN 為符記名稱。
- p->value.funptr = funs[i].ftype;
- 將第 i 個元素的 ftype 設定給 p 所指節點 value 同位結構裡的欄位 funptr，其型態為「*funtype」，定義如下。
- typedef double (*funtype) (double);　/*定義函式型態*/

- 【執行】
-
- C:\plone\ch14> mfrun.bat <Enter>
- C:\plone\ch14>bison mfcalc.y
- C:\plone\ch14>gcc mfcalc.tab.c -o mfcalc.exe
- C:\plone\ch14>mfcalc.exe < mfcalc.txt
- 輸入資料檔"mfcalc.txt"內容如下：
- pi=3.1416
- s=sin(pi/6)
- m=1
- n=2
- a=3
- b=sqrt(a)
- c=exp(m)
- d=ln(n)
- e=exp(1)

- 透過bison的yyparse()逐一剖析結果如下：
- putsym() p=0x003D2430 sym=260 name=sin var=0.000000
- putsym() p=0x003D2490 sym=260 name=cos var=0.000000
- putsym() p=0x003D24D0 sym=260 name=atan var=0.000000
- putsym() p=0x003D2510 sym=260 name=ln var=0.000000
- putsym() p=0x003D2550 sym=260 name=exp var=0.000000
- putsym() p=0x003D2590 sym=260 name=sqrt var=0.000000
- putsym() p=0x003D25D0 sym=259 name=pi var=0.000000
- 3.141600
- putsym() p=0x003D2610 sym=259 name=s var=0.000000
- getsym() p=0x003D2430 sym=260 name=sin var=0.000000
- getsym() p=0x003D25D0 sym=259 name=pi var=3.141600
- 0.500001
- putsym() p=0x003D2650 sym=259 name=m var=0.000000
- 1.000000
- putsym() p=0x003D2690 sym=259 name=n var=0.000000
- 2.000000
- putsym() p=0x003D26D0 sym=259 name=a var=0.000000
- 3.000000

- putsym() p=0x003D2710 sym=259 name=b var=0.000000
- getsym() p=0x003D2590 sym=260 name=sqrt var=0.000000
- getsym() p=0x003D26D0 sym=259 name=a var=3.000000
- 1.732051
- putsym() p=0x003D2750 sym=259 name=c var=0.000000
- getsym() p=0x003D2550 sym=260 name=exp var=0.000000
- getsym() p=0x003D2650 sym=259 name=m var=1.000000
- 2.718282
- putsym() p=0x003D2790 sym=259 name=d var=0.000000
- getsym() p=0x003D2510 sym=260 name=ln var=0.000000
- getsym() p=0x003D2690 sym=259 name=n var=2.000000
- 0.693147
- putsym() p=0x003D27D0 sym=259 name=e var=0.000000
- getsym() p=0x003D2550 sym=260 name=exp var=0.000000
- 2.718282

- ===== nodestackTop=0x003D27D0 nodesize=56 =====
- p=0x003D27D0 sym=259 name=e var=2.718282 next=0x003D2790
- p=0x003D2790 sym=259 name=d var=0.693147 next=0x003D2750
- p=0x003D2750 sym=259 name=c var=2.718282 next=0x003D2710
- p=0x003D2710 sym=259 name=b var=1.732051 next=0x003D26D0
- p=0x003D26D0 sym=259 name=a var=3.000000 next=0x003D2690
- p=0x003D2690 sym=259 name=n var=2.000000 next=0x003D2650
- p=0x003D2650 sym=259 name=m var=1.000000 next=0x003D2610
- p=0x003D2610 sym=259 name=s var=0.500001 next=0x003D25D0
- p=0x003D25D0 sym=259 name=pi var=3.141600 next=0x003D2590
- p=0x003D2590 sym=260 name=sqrt var=0.000000 next=0x003D2550
- p=0x003D2550 sym=260 name=exp var=0.000000 next=0x003D2510
- p=0x003D2510 sym=260 name=ln var=0.000000 next=0x003D24D0
- p=0x003D24D0 sym=260 name=atan var=0.000000 next=0x003D2490
- p=0x003D2490 sym=260 name=cos var=0.000000 next=0x003D2430
- p=0x003D2430 sym=260 name=sin var=0.000000 next=0x00000000

從執行的結果可以看出各變數最後所構成的 VAR 堆疊如下圖所示。

nodestackTop

name	var
e	2.718282
d	0.693147
c	2.718282
b	1.732051
a	3.000000
n	2.000000
m	1.000000
s	0.500001
pi	3.141600
sqrt	0.000000
exp	0.000000
ln	0.000000
atan	0.000000
cos	0.000000
sin	0.000000