(110-2) 資工系開課調查

國立中山大學資訊工程學系110學年度第2學

	科目名稱(限修人數)	任課教師	必/選	Monday	Tuesday
	微積分(二)	陳坤志	必		
	C程式設計(二)	柯正雯	必		
	C程式設計實驗(二)	柯正雯	必		
10人	數位電子學(限70人)	郭可騰	必	5,6,7 EC1006	
	計算機組織	黃英哲	必		2,3,4 EC50
授	Python 程式設計	希家史提夫	選		6,7,8 EC50
20人	機率學	陳嘉平	必		
	高等程式設計與實作	楊昌彪	選		
10人	資料探勘	蔡崇煒	選		
	作業系統	江明朝	必		
英	Unix系統程式	希家史提夫	必	2,3,4 EC5012	
	電腦網路	賴威光	必	6,7,8 EC5012	
授	安全車載通訊系統	克拉迪	選		6,7,8 EC50
	數值方法導論與應用	程正傑	選		
	電子設計自動化暨測試演算法	李淑敏	選		
	編譯器製作	張玉盈	必		
	專題製作實驗(一)	程正傑	必		
	資訊安全	范俊逸	選		2,3,4 EC10
	超大型積體電路設計概論	蕭勝夫	選		2,3,4 EC903
	軟體工程	李宗南	選		
	網路應用程式設計	王友群	選		
	安全電子商務	徐瑞壕	選		
析					
英	Unix系統程式	希家史提夫	選	2,3,4 EC5012	
	系統晶片之軟硬體協同驗證	黃英哲	選	2,3,4 EC5026	
	生醫影像分析專題	正雯	選	2,3,4 EC9014	

【重要確認】本系110學年

國立中山大學資訊工程學系110學年度第1學期

	科目名稱	任課教師	必/選修	Monday	Tuesday
大一					
語授課	微積分(一)	程正傑	必修		
	C程式設計(一)<含跨院通識>	蔣依吾	必修		
	C程式設計實驗(一)	蔣依吾	必修		
	離散數學	范俊逸	必修		
大二					
	數位系統實驗	鄺獻榮	必修	2,3,4 圖資PC3	
	人工智慧導論<含跨院通識>	蔡崇煒	選修	5,6,7 工EC2002	
	資料結構	楊昌彪	必修	6,7,8 工EC5012	
	數位系統	鄺獻榮	必修		2,3,4 工EC1006
	基礎訊號處理	郭可騰	選修		5,6,7 工EC1006
	線性代數<含跨院通識>	陳嘉平	必修		
	SystemC與數位系統設計概論<含跨	陳坤志	選修		
	數位影像處理	柯正雯	選修		
	程序導向程式設計	江明朝	選修		
大三					
	機器學習導論	張雲南	選修	2,3,4 工EC5012	
	資訊工程論壇	合 授	必修	5 工EC5012	
	資訊人與智慧財產權	黃英哲	選修		2,3,4 工EC1005
	電腦圖學概論	李宗南	選修		2,3,4 工EC5000
語授課	物件導向程式設計	克拉迪	必修		6,7,8 工EC5012
	無線網際網路<含跨院通識>	王友群	選修		
	演算法	蔡崇煒	必修		
	組合語言與微處理機	張雲南	必修		
	組合語言與微處理機實驗	張雲南	必修		
四					
	無線行動網路	賴威光	選修	6,7,8 工EC5007	
	網路系統程式設計	林俊宏	選修		6,7,8 工EC5007
	硬體描述語言	蕭勝夫	選修		6,7,8 工EC9032-
	無線與行動通訊安全理論與實務	徐瑞壕	選修		
	網際網路資料庫	張玉盈	選修		
	專題製作實驗（二）	蔡崇煒	必修		
語課	嵌入式系統程式設計	希[illegible]史提夫	選修		

1、名詞解釋：

a) 高階語言(High-level language)：接近於人類使用的語言，文語法的程式語言。容易學習，但執行效率比低階語言差。它的特性是，不受電腦機種、系統的限制，同一種高階語言可使用在不同的電腦系統，可攜性高。以程序（procedure）方式來描述解決問題的步驟。又稱為程序導向語言或第三代程式語言

b) 低階語言(Low-level language)：較接近電腦執行的動作，行效率高，但不易學習。任何機器語言的指令都包括兩部分：運算元與運算碼。而常見的低階語言有機器語言及組合語言。

c)流程圖：流程圖可分為系統流程圖及程式流程圖兩類。系統流圖（system flow chart）用以描述整個工作系統中，各單位之間的作業關係。程式流程圖(program flow chart)以表示程式中的處理過程，是流程圖中較常用者，因此以介紹程式流程圖為主。

d) 可攜性（portability）：表示同一語言程式被各種不同電腦系統執行的正確性；可攜性高者表示，同一程式不需修改即可在各種不同電腦正確的執行。每個OS提供的System Call當然都不一樣，即使每家Unix like的都長很像甚至很多system call 如read、write、fork等等在使用者角度看都一樣，這就是程式可攜性。

2、程式語言的演進，電腦語語言可分為幾種？

a)機器語言
b)組合語言
c)高階語言
d)第四代語言
e)自然語言

3、續上題，各種電腦語言的特性為何?請簡要說明！

a)機器語言：用一連串的0與1來代表資料或指令，可直接輸入電腦執行。程式簡潔、佔用記憶體少，而且執行時間最短。每個指令都有其相對應的機器碼，相當不方便。程式不易偵錯、撰寫麻煩，容易產生錯誤。

b)組合語言：使用符號化的語言，將數字、字母或特殊符號用簡單的文字替代。相較機器語言而言，容易偵錯及除錯。不同系統的電腦使用不同的組合語言，缺乏共通性。適用於專業的電腦工程師，不適合一般用途。

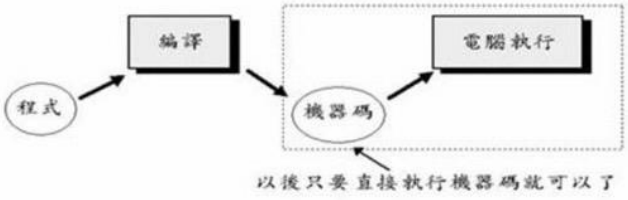
c)高階語言：具有機器無關（machine independence）的特性。在某一種電腦上所撰寫的高階語言程式，如果設計結構良好，通常不需要太多的修改就能挪到別種電腦上使用，這種特色又稱跨平台（cross platform）或可攜性（portability）。

d)第四代語言：第四代語言（The Fourth Generation和以往的程式語言大不相同，看起來是一串查詢需求的列表，這種表示方式很接近一般經理主管要求下屬提出報告的命令。第四代語言強調不需要太多的學習就能使用，和一般的電腦程式語言比起來，4GLS是比較友善的（user friendly）。

e)自然語言：自然語言（natural language）是人類文明的智慧結晶，也是電腦語言的終極目標，它的目標很單純，就是用一般的人類語言就能和電腦溝通。 自然語言的語法自由，經常一字多義，還視說話的人、事、物環境而有不同的詮釋，這些對電腦來說是相當困難的。 目前電腦科學家利用人工智慧方法來解決部份自然語言的問題，但也僅止於特殊的小範圍而已。

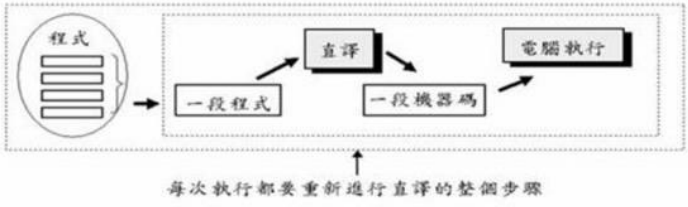
4、說明電腦語言之編譯方式: 1)編譯器，2)直譯器，請繪圖表示之！

4、說明電腦語言之編譯方式: 1)編譯器，2)直譯器，請繪圖表示之！



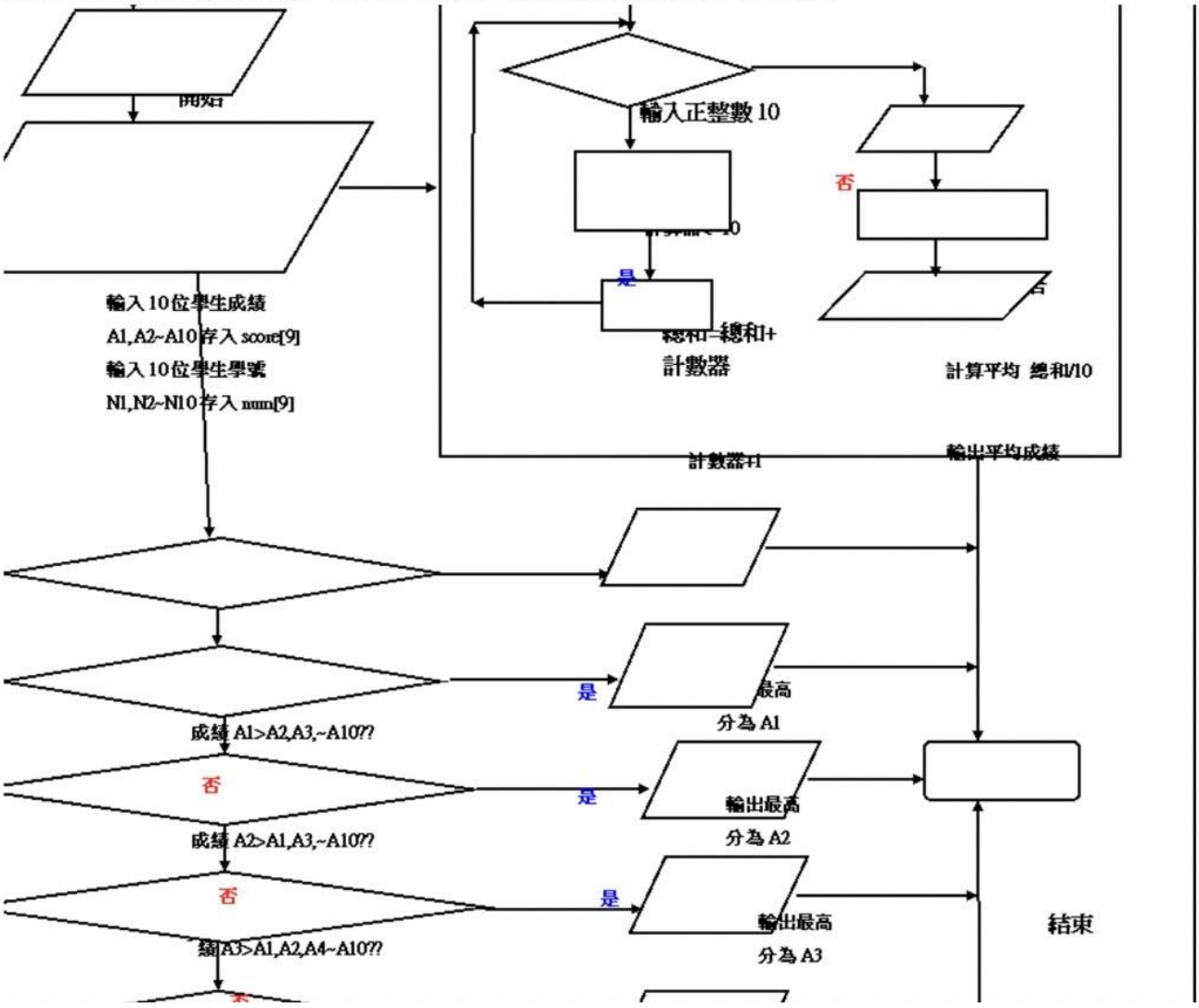
編譯器：編譯是先將程式全部翻譯成機器碼後，電腦再一口氣執行這些機器碼，以後再執行程式，只要執行機器碼，不需要再重新編譯。

直譯器：直譯則是每翻譯完一段程式，電腦就執行一段機器碼，接著再繼續翻譯下一段，電腦再執行一段機器碼，直到結束為止。每次電腦重新執行程式都要再經過直譯的過程。



5、請繪一流程圖，表示下列過程：

輸入十位同學的學號與成績，計算出平均值，將全班的最高分與平均值印出。



2.3.3. 編譯器

編譯器和直譯器兩者相比的話，有些不同，首先就是必須先把程式碼統統寫入到檔案裡面， 然後必須執行編譯器來試著編譯程式，如果編譯器不接受所寫的程式，那就必須一直修改程式， 直到編譯器接受且把你的程式編譯成執行檔。 此外，也可以在提示命令列，或在除錯器中執行你編譯好的程式看看它是否可以運作。

很明顯的，使用編譯器並不像直譯器般可以馬上得到結果。 不管如何，編譯器允許你作很多直譯器不可能或者是很難達到的事情。 例如：撰寫和作業系統密切互動的程式，甚至是你自己寫的作業系統！ 當你想要寫出高效率的程式時，編譯器便派上用場了。 編譯器可以在編譯時順便最佳化你的程式，但是直譯器卻不行。 而編譯器與直譯器最大的差別在於：當你想把你寫好的程式拿到另外一台機器上跑時， 你只要將編譯器編譯出來的可執行檔，拿到新機器上便可以執行， 而直譯器則必須要求新機器上，必須要有跟另一台機器上相同的直譯器， 才能組譯執行你的程式！

編譯式的程式語言包含 Pascal、C 和 c++， C 和 c++ 不是一個親和力十足的語言，但是很適合具有經驗的 Programmer。 Pascal 其實是一個設計用來教學用的程式語言，而且也很適合用來入門， 預設並沒有把 Pascal 整合進 base system 中， 但是 GNU Pascal Compiler 和 Free Pascal Compiler 都可分別在 lang/gpc 和 lang/fpc 中找到。

如果你用不同的程式來寫編譯式程式，那麼不斷地編輯-編譯-執行-除錯的這個循環肯定會很煩人，為了更簡化、方便程式開發流程，很多商業編譯器廠商開始發展所謂的 IDE (Integrated Development Environments) 開發環境， FreeBSD 預設並沒有把 IDE 整合進 base system 中， 但是你可透過 devel/kdevelop 安裝 kdevelop 或使用 Emacs 來體驗 IDE 開發環境。 在後面的 Using Emacs as a Development Environment 專題將介紹，如何以 Emacs 來作為 IDE 開發環境。

2.4. 用 cc 來編譯程式

本章範例只有針對 GNU C compiler 和 GNU C++ compiler 作說明， 這兩個在 FreeBSD base system 中就有了， 直接打 cc 或 gcc 就可以執行。 至於，如何用直譯器產生程式的說明，通常可在直譯器的文件或線上文件找到說明，因此不再贅述。

當你寫完你的傑作後，接下來便是讓這個程式可以在 FreeBSD 上執行， 通常這些要一些步驟才能完成，有些步驟則需要不同程式來完成。

```
OtherLetter     [A-DF-Za-df-z]
Letter          {E} | {OtherLetter}
IntLit          {Digit}+
%%
[ \t\n]+                                    { /* delete */
[Bb][Ee][Gg][Ii][Nn]                        { minor=0;   return(4);
[Ee][Nn][Dd]                                { minor=0;   return(5);
[Rr][Ee][Aa][Dd]                            { minor=0;   return(6);
[Ww][Rr][Ii][Tt][Ee]                        { minor=0;   return(7);
{Letter}({Letter} | {Digit} | _)*           { minor=0;   return(1);
{IntLit}                                    { minor=1;   return(2);
({IntLit}[.]{IntLit})({E}[+-]?{IntLit})?    { minor=2;   return(2);
\"([^\"\n] | \"\")*\"                       { stripquotes();
                                              minor=3;   return(2);
\"([^\"\n] | \"\")*\n                       { stripquotes();
                                              minor=0;   return(3);
"("                                         { minor=0;   return(8);
")"                                         { minor=0;   return(9);
";"                                         { minor=0;   return(10);
","                                         { minor=0;   return(11);
":="                                        { minor=0;   return(12);
"+"                                         { minor=0;   return(13);
" -"                                        { minor=0;   return(14);
%%
/* Strip unwanted quotes from string in yytext;    adjust yyleng. */
void stripquotes(void)
{
    Int frompos, topos = 0, numquotes = 2;

    for (frompos = 1; frompos < yyleng ; frompos++) {
        yytext[topos++] = yytext[frompos];
        if (yytext[frompos] == '"' && yytext[frompos+1] == '"') {
            frompos++;
            numquotes++;
        }
    }
    yyleng -= numquotes;
    yytext[yyleng] = '\0';
}
```

Figure 3.5 A Lex Definition for Extended Micro

直譯語言

此條目沒有列出任何參考或來源。
了解更多

直譯語言（英語：Interpreted language）是一種程式語言類型。這種類型的程式語言，會將程式碼一句一句直接執行，不需要像編譯語言（Compiled language）一樣，經過編譯器先行編譯為機器碼，之後再執行。這種程式語言需要利用直譯器，在執行期，動態將程式碼逐句直譯（interpret）為機器碼，或是已經預先編譯為機器碼的子程式，之後再執行。

理論上，任何程式語言都可以是編譯式，或直譯式的。它們之間的區別，僅與程式的應用有關。許多程式語言同時採用編譯器與直譯器來實作，其中包括Lisp，Pascal，BASIC 與 Python。JAVA及C#採用混合方式，先將程式碼編譯為位元組碼，在執行時再進行直譯。

直譯語言列表

- BASIC
- LISP
- Perl
- Python
- Ruby

理論上，任何程式語言都可以是編譯式，或直譯式的。它們之間的區別，僅與程式的應用有關。許多程式語言同時採用編譯器與直譯器來實作，其中包括Lisp，Pascal，BASIC 與 Python。JAVA及C#採用混合方式，先將程式碼編譯為位元組碼，在執行時再進行直譯。

直譯語言列表

- BASIC
- LISP
- Perl
- Python
- Ruby
- JavaScript
- PHP
- R

參見

- 手稿語言
- 編譯語言

HTML

用於創建網頁的標準標記語言

對於在維基百科上使用HTML，參見Help:HTML

超文本標記語言（英語：HyperText Markup Language，簡稱：HTML）是一種用於建立網頁的標準標記語言。HTML是一種基礎技術，常與CSS、JavaScript一起被眾多網站用於設計網頁、網頁應用程式以及行動應用程式的使用者介面[3]。網頁瀏覽器可以讀取HTML檔案，並將其彩現成視覺化網頁。HTML描述了一個網站的結構語意隨著線索的呈現，使之成為一種標記語言而非程式語言。

HTML元素是構建網站的基石。HTML允許嵌入圖像與物件，並且可以用於建立互動式表單，它被用來結構化資訊——例如標題、段落和列表等等，也可用來在一定程度上描述文件的外觀和語意。HTML的語言形式為尖括號包圍的HTML元素（如 `<html>` ），瀏覽器使用HTML標籤和指令碼來詮釋網頁內容，但不會將它們顯示在頁面上。

HTML可以嵌入如JavaScript的手稿語言，它們會影響HTML網頁的

副檔名	.html .htm
網路媒體型式	text/html
類型代碼（英語：Type code）	TEXT
開發者	W3C & WHATWG
初始版本	1993年，29年前

Implementation of Compilers

Instructor: Ye-In Chang (changyi@mail.cse.nsysu.edu.tw; 張玉盈， F5021)

Text book

1. Crafting a Compiler with C
   Charles N. Fischer and Richard J. Leblanc, Jr.
   The Benijamin/Cummings Publishing Company, 2005(開發)

Reference

1. Compilers : Principles, Techniques and Tools
   A. V. Aho, M. S. Lam, Ravi Sethi, and J. D. Ullman
   Addison-Wesley Publishing Company, 2nd ed., 2006 (台北書局).
2. System Software
   L. L. Beck
   Addison-Wesley Publishing Company, 3rd ed., 1997 (台北書局).
3. Lex & YACC 中譯本
   林偉豪 譯
   O'REILLY (美商歐萊禮), 1999

Grading

Lab.	30%
Midterm	30%
Final	40%

Topics (http://db.cse.nsysu.edu.tw/~changyi/slides/course.html)

(1) Introduction to Compiler
(2) Lexical Analysis
(3) Parsing, Grammar
(4) Semantic Analysis
(5) Code Optimization
(6) Storage Allocation
(7) Code Generation
(8) Lex and Yacc

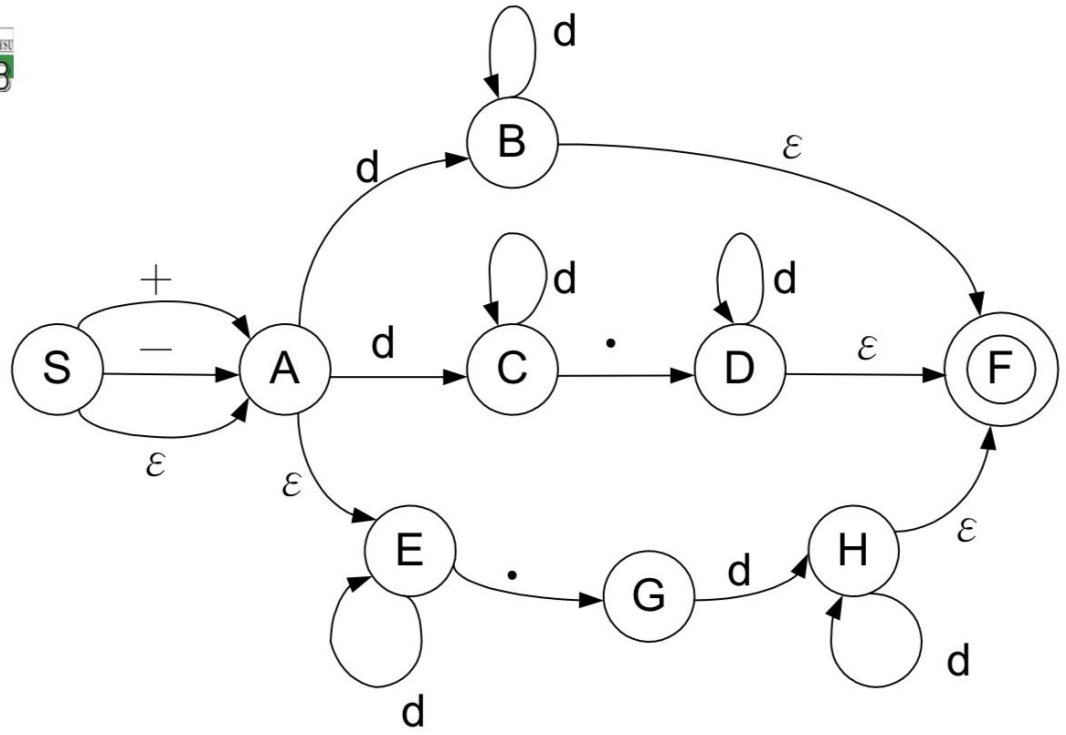


Figure 3.3. A non-deterministic machine equivalent to the machine in figure 3.1.

(+|-|ε) (d+|d+.d*|d*.d+) ➔ signed real number

		Input Symbols				
	δ	+	−	•	d	ε
	S	A	A			A
	A				B, C	E
	B				B	F
	C			D	C	
States	D				D	F
	E			G	E	
	F					
	G				H	
	H				H	F

transited to State B, not finial state.

Regular Grammar A->aB

2. State A, input a terminal symbol ‘a’ and is transited to State B, Finial state.

Regular Grammar A->a,A->aB

3.State A is both an initial State and a Finial State.

Regular Grammar A-> λ

有限自動機轉換正規文法

Example

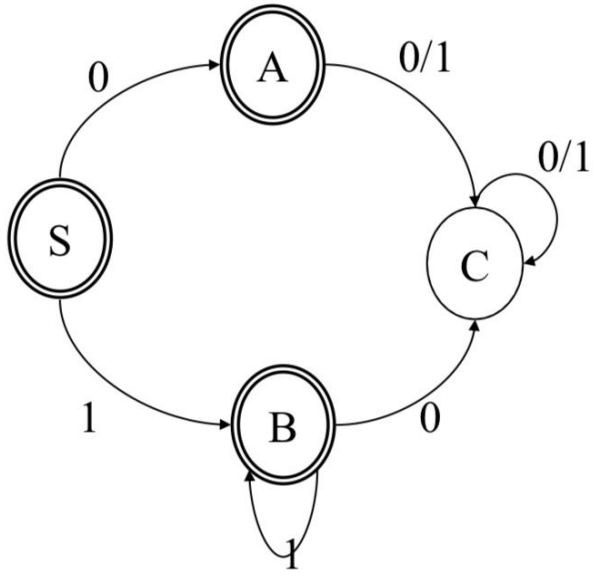
G=<N,T,S,P>,

N={S,A,B,C},

T={0,1},

P={

S-> λ, S->0, S->0A, S->1,S->1B,
A->0C, A->1C,
B->0C, B->1B, B->1,
C->0C, C->1C
}



- 文法範例

  G=<N,T, ∑,P>

  N={A,B, ∑}

  T={0,1}

  P={ ∑→A1,
  A→B0,
  B→1,
  B→B1 }

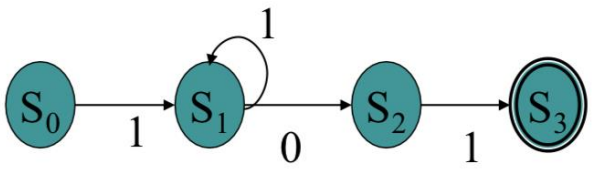
∑→A1

→B01→(B1)01 →(B1)101

→B1^{n}01→11^{n}01

→1^{n+1}01 →1$^+$01

即 {1$^+$01}={101,1101,11101,...}

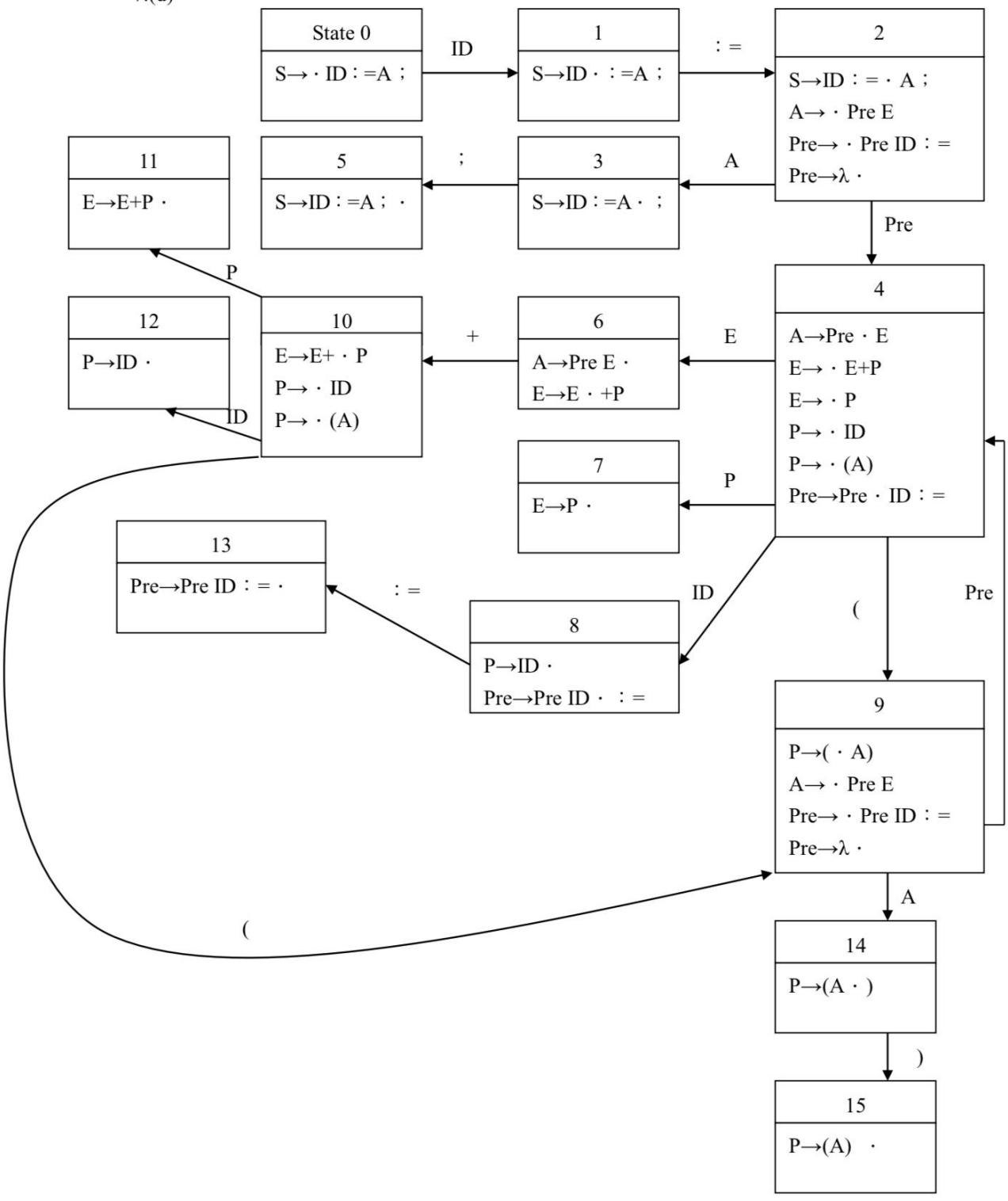


(FSM)

文法與機器之關係

語言	文法型態	相對應之機器	產生規則	
Type 0	Unrestricted grammar	Turing machine	$\alpha\rightarrow\beta$, $\alpha\neq\lambda$ $\alpha,\beta\in(N\cup T)^*$	
Type 1	Context sensitive Grammar	Linear bounded automata	$\gamma_1\alpha\gamma_2\rightarrow\gamma_1\beta\gamma_2$ $\gamma_1,\gamma_2\in(N\cup T)^*$ $\alpha\in N$, $\beta\in(N\cup T)^*-\lambda$, length($\alpha$)≦ length($\beta$)	
Type 2	Context free grammar	Pushdown automata	$A\rightarrow\beta$, $\beta\in(N\cup T)^*-\lambda$, $A\in N$	
Type 3	Regular grammar	Finite automata or FSM	$A\rightarrow aB$, $A\rightarrow a$, $A,B\in N$, $a\in T$, (right-linear)	$A\rightarrow Ba$, $A\rightarrow a$, $A,B\in N$, $a\in T$, (left-linear)

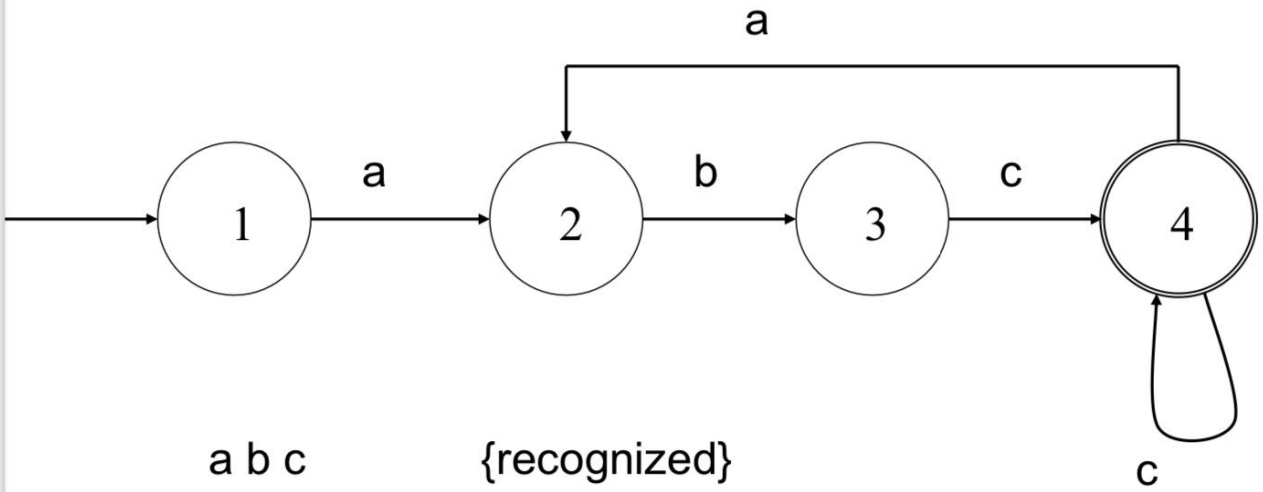
7.(d)



Scanner

<<Scanner1.ppt>>

Modeling Scanners as Finite Automata



a b c {recognized}
a b c c a b c {recognized}
a c { not recognized}

Graphical representation of a finite automaton

Language: set of legal tokens.

Legal token: defined by Grammer.

Machine(ex. FSM): recognize legal tokens.

Yacc 範例檔 & Java測試檔
Java之Lex/Yacc Lab.之文法
Java之Lex/Yacc Lab.之文法 (Provided by the student at 2021.06.09)

Lab. 2022

(HW1作業說明)Simple Pascal Scanner
(HW1) Scanner測資 & 範例檔
(HW1) 教學與作業注意事項 (pdf)
去年Lex_FAQ參考

(HW2作業說明)Simple Pascal Parser
(HW2) Parser測資 & 範例檔
yacc補充的範例檔(5/28)
Yacc 教學與作業注意事項 (pdf)
Pascal之Lex/Yacc Lab.之文法(參考用)

References

Lex & Yacc 的用法
Lex & Yacc 的來源
flex user manual (ps file) (pdf)
bison user manual (ps file)(pdf)
lex user manual (ps file) (pdf)
yacc user manual (ps file) (pdf
Lexer and Parser Generators (a link)
A Compact Guide to Lex & Yacc (a link)
Function+Procedure

< 上一頁 >

Chapter 1 Introduction

Outlines

- 1.1 Overview and History
- 1.2 What Do Compilers Do?
- 1.3 The Structure of a Compiler
- 1.4 The Syntax and Semantics of Programming Languages
- 1.5 Compiler Design and Programming Language Design
- 1.6 Compiler Classifications
- 1.7 Influences on Computer Design

(7) Code Generation	Code Generation.doc (ppt) (pdf) Introd3.doc (ppt) (pdf) Introd4.doc (ppt) (pdf) LoaderLinker.doc (ppt)(pdf)
(8) Lex and Yacc	Lex and Yacc.doc (ppt) (pdf) An Example.ppt (pdf)

Textbook

Chapter1 (pdf)
Chapter2 (pdf)
Chapter3 (pdf), (Fig. 3.5) (pdf)
Chapter4 (pdf)
Chapter5 (pdf)
Chapter6 (pdf)
Chapter7 (pdf)
Chapter8 (pdf)
Lex_note (pdf)
Lex program example
Yacc program example

教學大綱

syllabus2022

參考用書

- Crafting a Compiler with C, The Benijamin/Cummings Publishing Company (開發), 2005
- Compilers : Principles, Techniques and Tools, 2nd ed., Addison–Wesley Publishing Company (台北書局), 2006
- System Software,3rd ed., Addison–Wesley Publishing Company (台北書局), 1997
- Lex & YACC 中譯本, O'REILLY (美商歐萊禮), 1999

Lab. 2021

Lex 評分標準與Demo項目 (pdf)
Lex 教學與作業注意事項 (pdf)
(HW1 作業說明) Lex Homework Simple Java – Scanner (pdf)
Lex Java測試檔
Lex相關提醒
去年Lex_FAQ參考

(HW2 作業說明) Yacc Homework Simple Java – Parser (pdf)
Yacc 教學與作業注意事項 (pdf)
Yacc 範例檔 & Java測試檔
Java之Lex/Yacc Lab.之文法
Java之Lex/Yacc Lab.之文法 (Provided by the student at 2021.06.09)

Lab. 2022

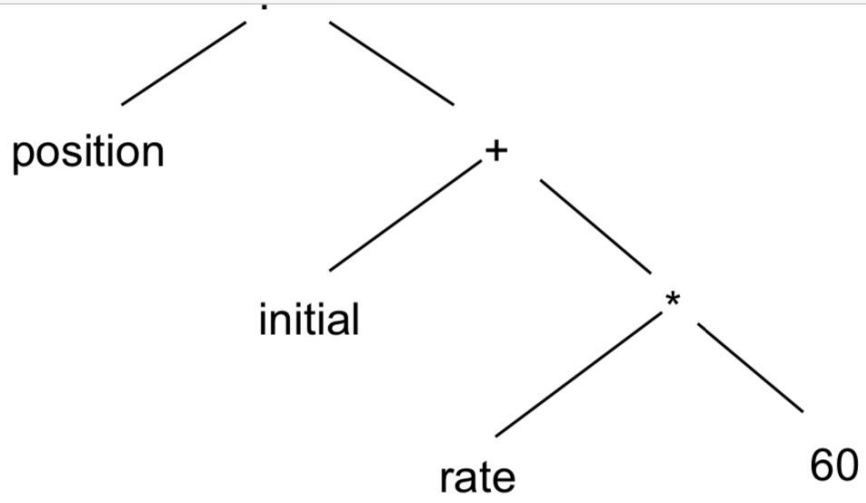
轉譯程式(Translator)

- 為一系統軟體,其功能是將輸入的原始程式(Source Program)轉換成另一種相對應的程式語言(如組合語言 機器語言)
- 包含下列四種
  - Assembler
  - Compiler
  - Preprocessor
  - Interpreter

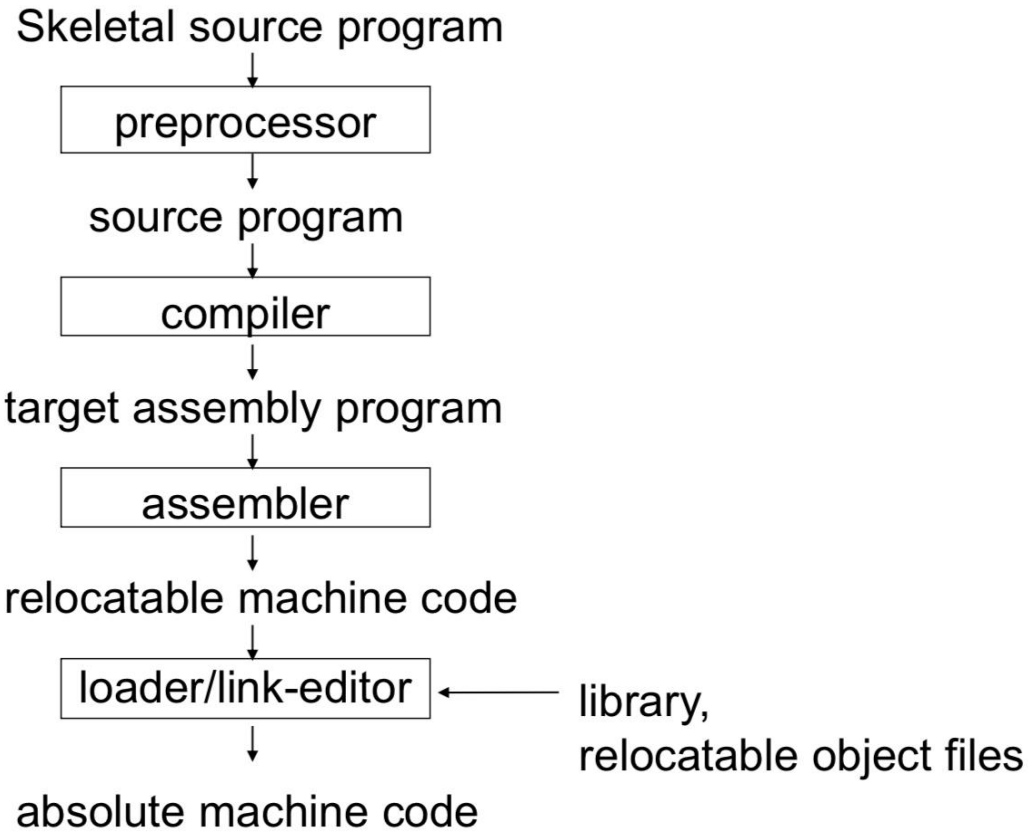
編譯程式

1.語彙分析階段(Lexical Analysis Phase)

- 變數 常數 區分符號 關鍵字 運算元
- 文字表(Literal Table)
- 識別字表(Identifier Table)

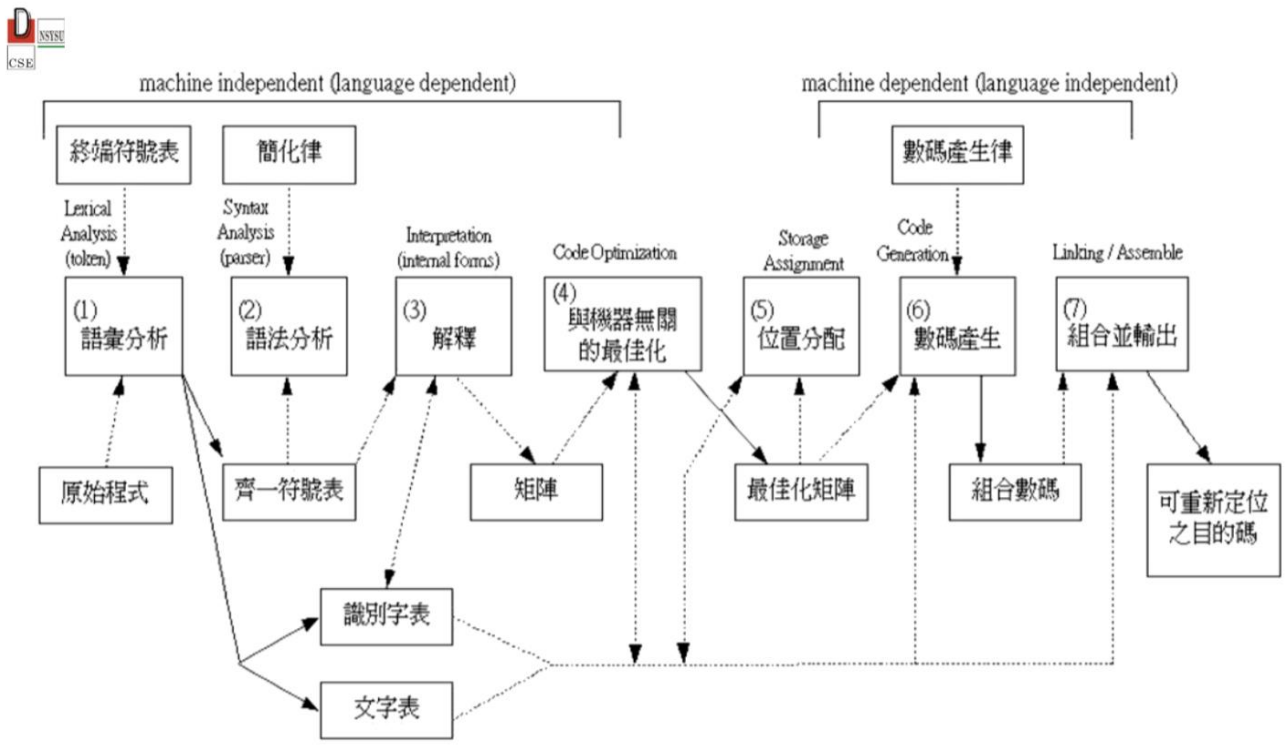


Syntax tree for position:=initial + rate* 60



A language-processing system

- 簡介四個“與機器無關的最佳化”技巧如下：
  - 刪除共同的副式子(common subexpression)。
  - 將常數(constants)間的運算先行計算。
  - 布林式子Boolean expression)的最佳化。
  - 將迴路(loop)中不變的計算式子移出迴路外。



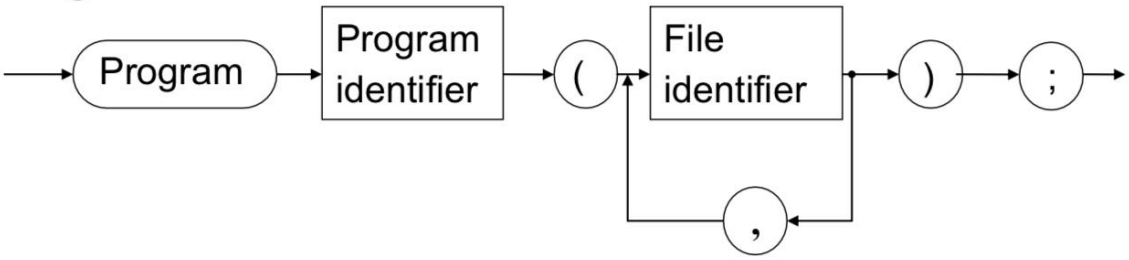
編譯程式的結構圖

Pascal

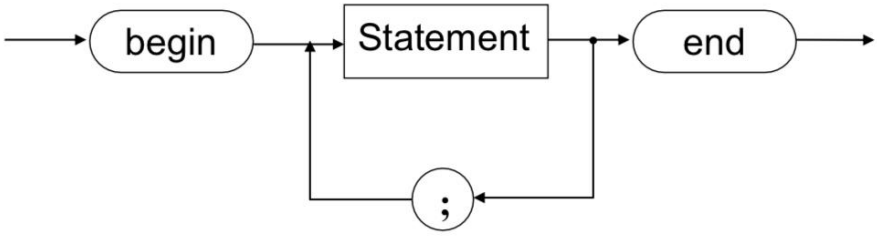
<<Pascal1.ppt>>

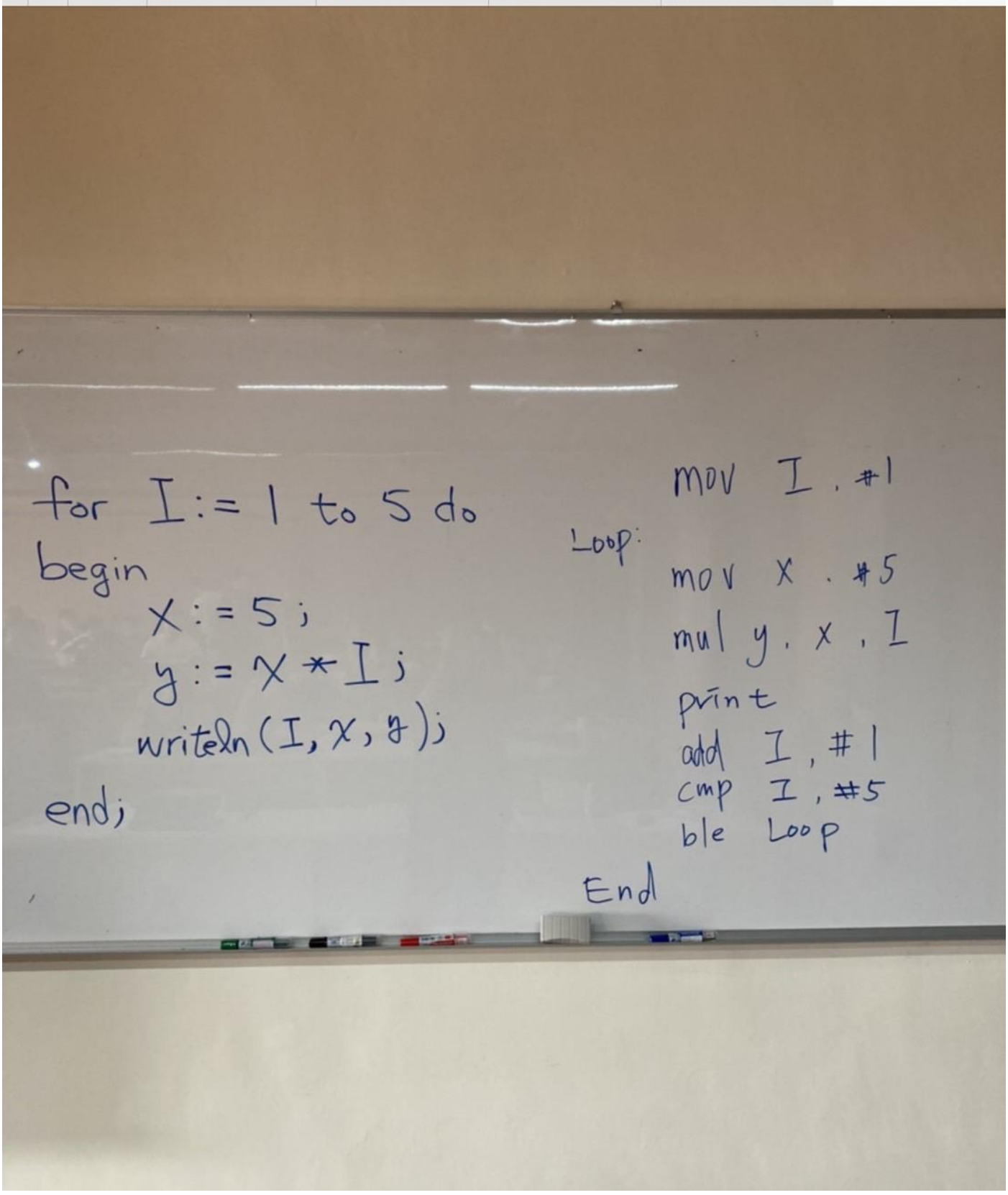
Program

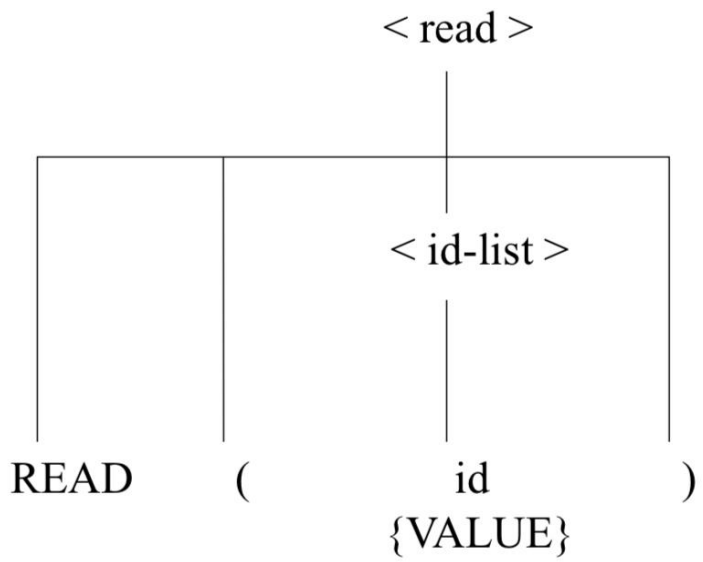
Program header



Process part



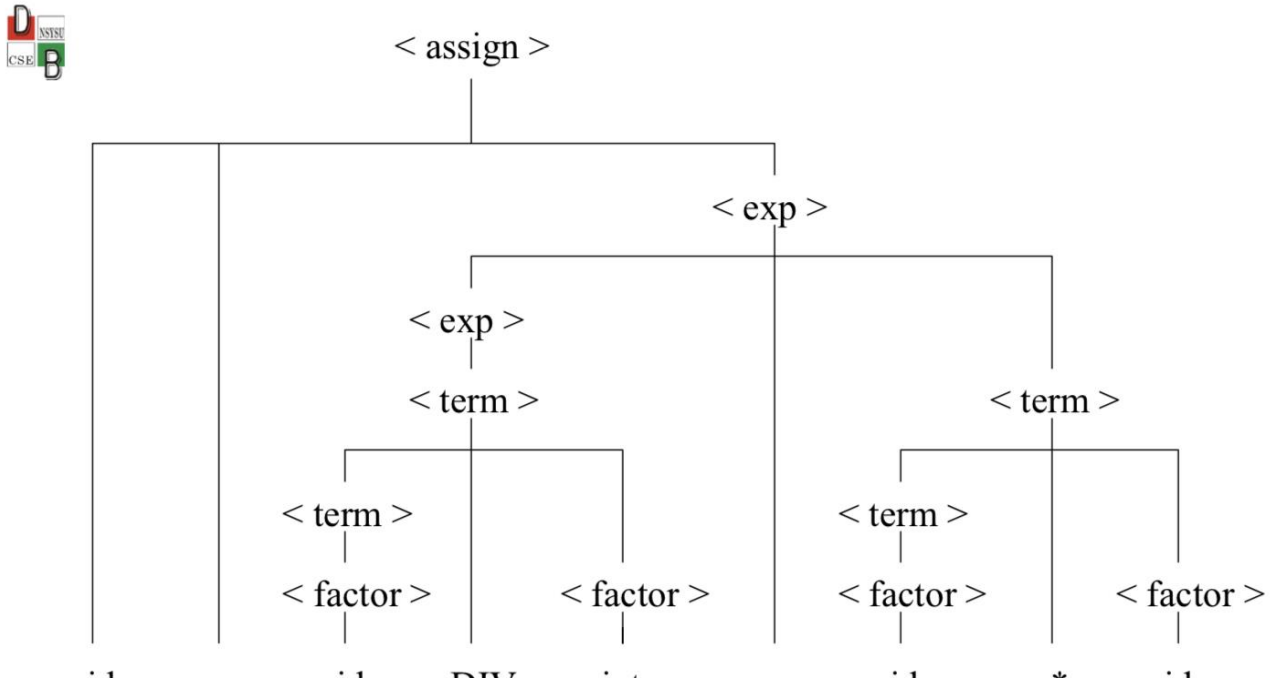




(a)

FIGURE 5.3 Parse trees for two statements from Fig. 5.1.

<<ParseTree.ptt>>



Grammar for Micro with Action Symbols

```
<program>        → #start begin <statement list> end
<statement list> → <statement> {<statement>}
<statement>      → <ident> := <expression> #assign;
<statement>      → read (<id list>);
<statement>      → write (<expr list>);
<id list>        → <ident>#read_id{,<ident>#read_id}
<expr list>      → <expression>#write_
                    expr{,<expression>#write_expr}
<expression>     → <primary>{<add op> <primary>
                    #gen_infix}
```


```
<primary>        → (<expression>)
<primary>        → <ident>
<primary>        → INTLITERAL #process_literal
<add op>         → PLUSOP #process_op
<add op>         → MINUSOP #process_op
<ident>          → ID #process_id
<system goal>    → <program> SCANEOF #finish
```

Simplified Pascal grammar. (Extended BNF grammar)

```
1<prog>     ::= PROGRAM <prog-name>
                VAR <dec-list> BEGIN <stmt-list> END.
2<prog-name> ::= id
3a<dec-list> ::= <dec>  {; <dec>}
4<dec>       ::= <id-list> : <type>
5<type>      ::= INTEGER
6a<id-list>  ::= id {, id}
7a<stmt-list>::= <stmt> {; <stmt>}
8<stmt>      ::= <assign> | <read> | <write> | <for>
9<assign>    ::= id := <exp>
```


Simplified Pascal grammar. (Extended BNF grammar)

```
10a<exp>    ::= <term> {+<term>| - <term>}
11a<term>   ::= <factor> { * <factor> |DIV<factor> }
12<factor> ::= id | int | ( <exp> )
13<read>    ::= READ ( <id-list> )
14<write>   ::= WRITE ( <id-list> )
15<for>     ::= FOR <index-exp> DO <body>
16<index-exp>    ::= id := <exp> TO <exp>
17<body>    ::= <stmt> | BEGIN <stmt-list> END
```

Simplified Pascal grammar. (BNF grammar)

```
1<prog>     ::= PROGRAM <prog-name> VAR <dec-list>
                BEGIN <stmt-list> END.
2<prog-name>     ::= id
3<dec-list>::= <dec> | <dec-list> ; <dec>
4<dec>      ::= <id-list> : <type>
5<type>     ::= INTEGER
6<id-list> ::= id | <id-list>, id
7<stmt-list>     ::= <stmt> | <stmt-list> ; <stmt>
8<stmt>     ::= <assign> | <read> | <write> | <for>
9<assign>   ::= id := <exp>
```


Simplified Pascal grammar. (BNF grammar)

```
10<exp>     ::= <term> | <exp> + <term> |
                <exp> - <term>
11<term>    ::= <factor> | <term> * <factor> |
                <term> DIV <factor>
12<factor> ::= id | int | ( <exp> )
13<read>    ::= READ ( <id-list> )
14<write>   ::= WRITE ( <id-list> )
15<for>     ::= FOR <index-exp> DO <body>
16<index-exp>     ::= id := <exp> TO <exp>
17<body>    ::= <stmt> | BEGIN <stmt-list> END
```

控制結構

Pascal是結構化編程語言，意味著控制流被結構化成標準語句，理想地沒有「go to」命令。

```
while a <> b do writeln('Waiting');

if a > b then writeln('Condition met')
else writeln('Condition not met');

for i := 1 to 10 do writeln('Iteration: ',
i:1);

repeat
```

控制結構

Pascal是結構化編程語言，意味著控制流被結構化成標準語句，理想地沒有「go to」命令。

```
while a <> b do writeln('Waiting');

if a > b then writeln('Condition met')
else writeln('Condition not met');

for i := 1 to 10 do writeln('Iteration: ',
i:1);

repeat
  a := a + 1
until a = 10;

case i of
  0: write('zero');
  1: write('one');
  2: write('two')
end;
```

過程和函式

Pascal將程式結構化成過程和函式。

Pascal 的版本演進

雖然 Pascal 有國際標準，但 Pascal 的標準對使用 Pascal 來說不太重要。所以，本節的內容只是給讀者了解一下 Pascal 的發展歷史。

早期的發展

Pascal 是瑞士電腦科學家 Niklaus Wirth 教授所設計和開發的程式語言，其目的是做為教學工具。Niklaus Wirth 教授在 *Algorithms + Data Structures = Programs* 這本教科書的第一版使用 Pascal 來寫範例程式碼。這本書的書名也成了資訊界的經典金句。

由於 Pascal 一開始尚未標準化，理所當然以 Niklaus Wirth 教授所設計的 Pascal 編譯器為準。另外，早期的 Mac 系統曾經使用 Pascal 做為其應用程式語言。

Pascal

雖然 Pascal 有國際標準 (ISO 7185)，但該標準本身設計不良，故現行的 Pascal 編譯器很少遵守這個版本的 Pascal 方言，而會加入其他的特性。

Extended Pascal

保留字 (Keywords)

保留字 (keywords 或 reserved words) 在程式碼中有特別的意義，不能做為識別字。以下是各個 Pascal 方言的保留字：

- Delphi 的保留字清單
- Free Pascal 的保留字清單

由於 Pascal 保留了 Algol 的慣例，以英文單字當成 (一部分的) 運算子和設置程式碼區塊，故保留字會比較多。剛從 C 語言轉換過來的程式設計者可能會感到不太適應。持續寫一段時間的 Pascal 程式就會習慣了。

對於剛學程式設計的讀者來說，可能會覺得保留字難以記憶。實際上，程式設計師不會去記憶保留字，而會在學習程式設計的過程中自然而然地學會保留字。此外，編輯器會以顏色來提示程式設計者保留字出現的位置。

鎖定 Pascal 方言 (Dialect)

在使用 Free Pascal 時，可以鎖定 Pascal 方言。以下是目前在 Free Pascal 中可用的方言：

- `fpc` : 預設模式

2.3.3. 編譯器

編譯器和直譯器兩者相比的話，有些不同，首先就是必須先把程式碼統統寫入到檔案裡面， 然後必須執行編譯器來試著編譯程式，如果編譯器不接受所寫的程式，那就必須一直修改程式， 直到編譯器接受且把你的程式編譯成執行檔。 此外，也可以在提示命令列，或在除錯器中執行你編譯好的程式看看它是否可以運作。

很明顯的，使用編譯器並不像直譯器般可以馬上得到結果。 不管如何，編譯器允許你作很多直譯器不可能或者是很難達到的事情。 例如：撰寫和作業系統密切互動的程式，甚至是你自己寫的作業系統！ 當你想要寫出高效率的程式時，編譯器便派上用場了。 編譯器可以在編譯時順便最佳化你的程式，但是直譯器卻不行。 而編譯器與直譯器最大的差別在於：當你想把你寫好的程式拿到另外一台機器上跑時， 你只要將編譯器編譯出來的可執行檔，拿到新機器上便可以執行， 而直譯器則必須要求新機器上，必須要有跟另一台機器上相同的直譯器， 才能組譯執行你的程式！

編譯式的程式語言包含 Pascal、C 和 c++， C 和 c++ 不是一個親和力十足的語言，但是很適合具有經驗的 Programmer。 Pascal 其實是一個設計用來教學用的程式語言，而且也很適合用來入門， 預設並沒有把 Pascal 整合進 base system 中， 但是 GNU Pascal Compiler 和 Free Pascal Compiler 都可分別在 lang/gpc 和 lang/fpc 中找到。

如果你用不同的程式來寫編譯式程式，那麼不斷地編輯-編譯-執行-除錯的這個循環肯定會很煩人，為了更簡化、方便程式開發流程，很多商業編譯器廠商開始發展所謂的 IDE (Integrated Development Environments) 開發環境， FreeBSD 預設並沒有把 IDE 整合進 base system 中， 但是你可透過 devel/kdevelop 安裝 kdevelop 或使用 Emacs 來體驗 IDE 開發環境。 在後面的 Using Emacs as a Development Environment 專題將介紹，如何以 Emacs 來作為 IDE 開發環境。

2.4. 用 cc 來編譯程式

本章範例只有針對 GNU C compiler 和 GNU C++ compiler 作說明， 這兩個在 FreeBSD base system 中就有了， 直接打 cc 或 gcc 就可以執行。 至於，如何用直譯器產生程式的說明，通常可在直譯器的文件或線上文件找到說明，因此不再贅述。

當你寫完你的傑作後，接下來便是讓這個程式可以在 FreeBSD 上執行， 通常這些要一些步驟才能完成，有些步驟則需要不同程式來完成。

```
for(int i = 0; i < 10
    printf("%d ", i);
}

return 0;
```

來看個簡單的範例：

```c
#include <stdio.h>

int main(void) {
    for(int i = 0; i < 10; i++) {
        printf("%d ", i);
    }

    return 0;
}
```

執行結果：

```
0 1 2 3 4 5 6 7 8 9
```

簡單的例子，但說明 for 的作用再適合不過，在
中宣告變數與指定初始值，這個宣告的變數在 f

```
while(score != -1) {
        count++;
        sum += score;
        printf("輸入分數(-1結束)
        scanf("%d", &score);
}
```

一個計算輸入成績平均的程式如下所示：

```c
#include <stdio.h>

int main(void) {
    int score = 0;
    int sum = 0;
    int count = -1;

    while(score != -1) {
        count++;
        sum += score;
        printf("輸入分數(-1結束)：");
        scanf("%d", &score);
    }

    printf("平均：%f\n", (double) sum / count );

    return 0;
}
```

執行結果：

2.3.1. 直譯器

使用直譯器時，所使用的程式語言就像變成一個會和你互動的環境。 當在命令提示列上打上命令時，直譯器會即時執行該命令。 在比較複雜的程式中，可以把所有想下達的命令統統輸入到某檔案裡面去， 然後呼叫直譯器去讀取該檔案，並且執行你寫在這個檔案中的指令。 如果所下的指令有錯誤產生，大多數的直譯器會進入偵錯模式(debugger)， 並且顯示相關錯誤訊息，以便對程式除錯。

這種方式好處在於：可以立刻看到指令的執行結果，以及錯誤也可迅速修正。 相對的，最大的壞處便是當你想把你寫的程式分享給其他人時，這些人必須要有跟你一樣的直譯器。 而且別忘了，他們也要會使用直譯器直譯程式才行。 當然使用者也不希望不小心按錯鍵，就進入偵錯模式而不知所措。 就執行效率而言，直譯器會使用到很多的記憶體， 而且這類直譯式程式，通常並不會比編譯器所編譯的程式的更有效率。

筆者個人認為，如果你之前沒有學過任何程式語言，最好先學學習直譯式語言(interpreted languages)， 像是 Lisp，Smalltalk，Perl 和 Basic 都是， 的 shell 像是 `sh` 和 `csh` 它們本身就是直譯器，事實上，很多人都在它們自己機器上撰寫各式的 shell "script"， 來順利完成各項 "housekeeping(維護)" 任務。 的使用哲學之一就是提供大量的小工具， 並使用 shell script 來組合運用這些小工具，以便工作更有效率。

2.3.2. FreeBSD 提供的直譯器

下面這邊有份 Ports Collection 所提供的直譯器清單，還有討論一些比較受歡迎的直譯式語言

至於如何使用 Ports Collection 安裝的說明，可參閱 FreeBSD Handbook 中的 Ports章節。

BASIC

> BASIC 是 Beginner's ALL-purpose Symbolic Instruction Code 的縮寫。 BASIC 於 1950 年代開始發展，最初開發這套語言的目的是為了教導當時的大學學生如何寫程式。 到了 1980，BASIC已經是很多 programmer 第一個學習的程式語言了。 此外，BASIC 也是 Visual Basic 的基礎。
>
> FreeBSD Ports Collection 也有收錄相關的 BASIC 直譯器。 Bywater Basic 直譯器放在 lang/bwbasic。 而 Phil Cockroft's Basic 直譯器(早期也叫 Rabbit Basic)放在 lang/pbasic。

Lisp