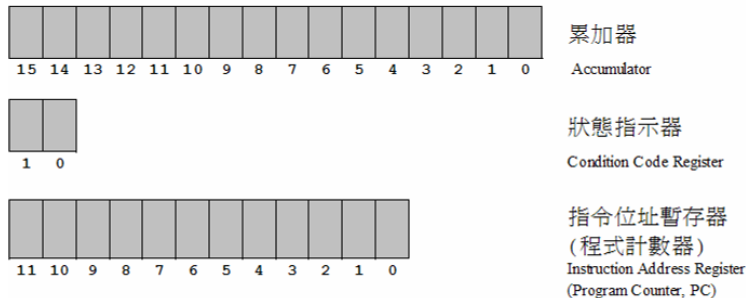
系統程式

From 線上測驗網站 http://david.bioinformatic.idv.tw/LawQuiz/

第9章. 組譯程式

9.1. 基本概念

9.1.1.中央處理單元內部有三個元件



一 累加器(Accumulator)

擔任運算工作的元件，一個運算基本上需要兩個步驟完成：

- 將運算元載入(Load)累加器
- 與另一運算元執行運算（加、減、乘、除、數值比較）運算結果置於累加器（數值比較則改變狀態指示器之值）

二 狀態指示器(Condition code register)

表示累加器進行數值比較結果的狀態

三 指令位址暫存器(Instruction address register)

- 又稱程式計數器(Program counter, location counter)
- 將擷取之下一個指令之記憶體位址

9.1.2.運算碼的設定

運算碼	助憶詞		動作
0001 = 1	LDA	載入	ACCU ← MM(LOC)
0010 = 2	STA	儲存	MM(LOC) ← ACCU
0011 = 3	ADD	加法運算	ACCU ← ACCU + MM(LOC)
0100 = 4	SUB	減法運算	ACCU ← ACCU - MM(LOC)
0101 = 5	MUL	乘法運算	ACCU ← ACCU * MM(LOC)
0110 = 6	DIV	除法運算	ACCU ← ACCU / MM(LOC)
0111 = 7	MOD	餘數運算	ACCU ← ACCU Mod MM(LOC)
1000 = 8	CMP	數値比較	ACCU :MM(LOC)
1001 = 9	JMP	跳離	PC ← LOC
1010 = A	JLT	値小跳離	If ACCU < MM(LOC') Then PC ← LOC
1011 = B	JEQ	等値跳離	If ACCU = MM(LOC') Then PC ← LOC
1100 = C	JGT	値大跳離	If ACCU > MM(LOC') Then PC ← LOC
1101 = D	IN	輸入	MM(LOC) ← input data
1110 = E	OUT	輸出	print MM(LOC)
1111 = F	HALT	終結	halt

9.1.3.機器週期(Machine Cycle)

讀取指令，解碼，讀取運算元，執行

一 機械語言指令(Machine Language Instruction)

每當我們從鍵盤鍵入一個命令、資料或用滑鼠從畫面中選取某個選項之後，電

腦系統為了執行此一命令或處理資料時，會先翻成電腦內部看得懂的機械語言指令(Machine Language Instruction)。

二　機器週期(Machine Cycle)

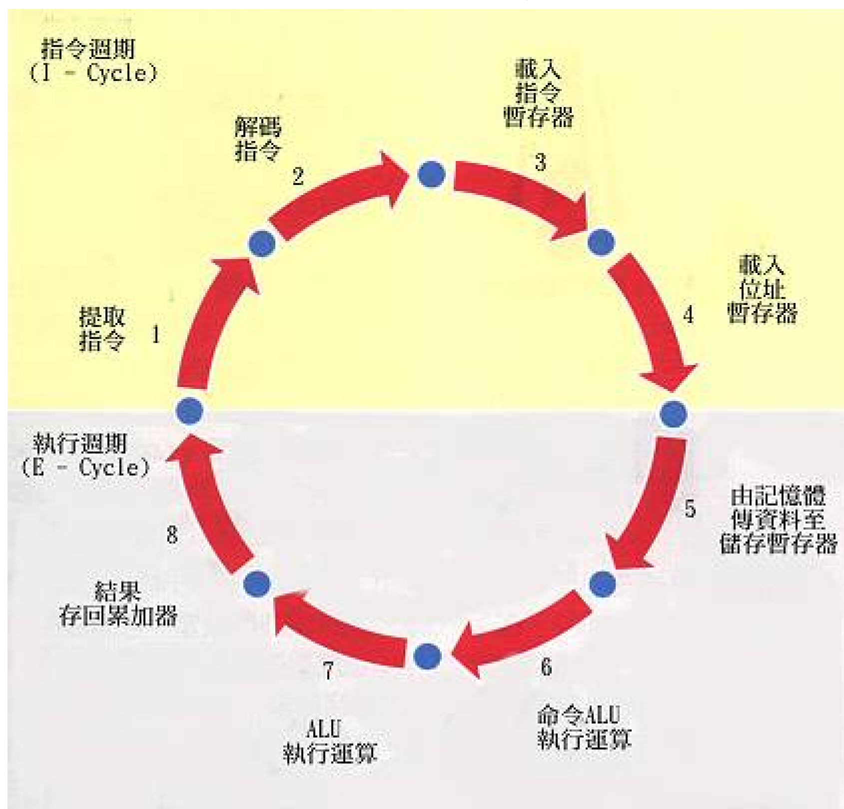
而處理每一個機械語言指令時，資料會傳送或來自記憶體或輸入/輸出單元。如此每一次的傳送或接收的動作稱為機器週期(Machine Cycle)。

三　指令週期與執行週期

機器週期又可分為兩部份：指令週期(Instruction cycle，I-cycle)與執行週期（Execution cycle，E-cycle）。

在指令週期中，控制單元會從記憶單元取出下一待執行的指令。在執行週期(E-cycle)內所執行的工作包含：找出資料、執行指令，以及將結果存到累加器內，現在我們用圖來表示之。

- **指令週期**
  - 控制單元從主記憶體中提取下一個所要執行的指令。
  - 控制單元從指令予以解碼(decode)
  - 控制單元將指令中用來說明要執行什麼動作的這一部份，存入指令暫存器
  - 控制單元將指令中用來說明相關資料所儲存之位置這一部份，存入位址暫存器。
- **執行週期**
  - 控制單元根據位址暫存器內的資訊，從主記憶體中讀啟所需的資料，並將其存入 ALU 的儲存暫存器內。
  - 控制單元根據指令暫存的資訊，命令 ALU 去執行所需的運算。
  - ALU 執行所需之運算，它會將發現於儲存暫存器以及累加器內的數值予以相加。
  - 運算結果再存回累加器內，這個動作會清除累加器原先所儲存的數值。



9.2. 組合語言的指令種類

9.2.1.助憶碼指令(mnemonic instruction)

具有三個欄位

- **標記(label)**
- **運算碼(Operation-code)**
  - 是電腦中央處理單元具有之各項功能指令的代碼，使用 4 個位元，代碼總數最多只有 16 個 (24) 。
- **運算元(operand)**
  - 可以說是運算碼的參數，依據個別指令的設定，這個參數可能是主記憶體的位址編號，以對應欲處理之資料，也可能直接對應為一個數值進行處理，有些運算碼可能不需要任何參數。（實際的電腦可能有些運算碼不只一個參數，即運算元可能多個！）

9.2.2.假指令(pseudo instruction)

常見的假指令有以下幾種

- **設定程式的開始處及結束處**
  - START:指定程式的名稱及起始位址
  - END:程式結束處
- **保留記憶體空間**
  - 保留記憶體空間並設定初值：BYTE2，代表保留一個 Byte 的記憶體空間，並設定其初值為 2
  - 保留記憶體空間但不設定初值：RESB2，代表保留兩個 Byte 的記憶體空間，但未設定初值

虛擬指令不會轉換成機器碼，而是用來指引組譯器。

9.2.3.巨集指令(macro instruction)

參考巨集處理程式一章

9.3. 組譯程式的工作

組譯程式的目的就是將原始程式處理成機器碼。而組譯程式必須執行的工作有以下五種

- 將助憶碼指令轉換成為相對應的機器碼
- 將符號運算元(symbolic operand)轉換成相對應的機器位址
- 以機器所能接受的格式產生機器指令
- 將資料常數(constant)轉換成機器內部的表示法(因為不同的機器數值資料的範圍、精準度可能會不同)
- 產生的目的碼(object code)及組合程式列表(組譯清單)

9.4. 目的碼的內容

9.4.1.H-record(標頭紀錄)

即 Head record。紀錄程式的名稱，程式的起始位址以及程式的長度等三種資訊。

9.4.2.T-record(文本紀錄)

即 Text record。紀錄程式的內容，包含目的碼起始的位址，指令的機器碼，欲載入的位址及資料

9.4.3.E-record(結束紀錄)

即 End record。紀錄程式的結束處。並可以指定程式第一個開始執行的指令的位址。

9.5. 組譯程式的種類

9.5.1.單次處理組譯程式

9.5.2.兩次處理組譯程式

一 Forward reference

程式稍後會定義 label 的參考

二 Two Pass 的進行步驟

第一次掃描原始程式的標籤定義和指定位址，第二次執行 forward reference

- ♦ **第一次執行(定義符號)**
  - 決定所有敘述的位址
  - 決定所有 label 的位址
  - 處理組譯器指令(假指令)，包括會影響位址指派的處理，例如決定 BYTE、RESW 等定義的資料區域長度
- ♦ **第二次執行(產生目的程式)**
  - 對所有指令產生目的程式
  - 產生 BYTE、WORD、DC、DS 及 literal 等定義的資料值
  - 執行第一次所沒執行的虛擬指令處理
  - 產生目的程式和組譯清單(Assembly Listing)

三 資料結構

Pass1:

- ♦ **輸入**
  - 原始程式碼
- ♦ **參考資料結構**
  - Location Counter(LC)
  - Machine Operation Table(OPTAB)
  - Pseudo Operation Table(POT)
- ♦ **輸出**
  - Symbol Table(ST)
  - Literal Table(LT)
  - 原始程式副本

Pass2:

- ♦ **輸入**
  - Symbol Table(ST)
  - Literal Table(LT)
  - 原始程式副本
- ♦ **參考資料結構**
  - Location Counter(LC)
  - Machine Operation Table(OPTAB)
  - Pseudo Operation Table(POT)
  - Base register table-BT
  - 工作區(Working space)
- ♦ **輸出**
  - 目的程式
  - 原始程式列表

9.5.3.多次處理組譯程式

```
ALPHA  EQU    BETA
BETA   EQU    DELTA
DELTA  RESW   1
```

組譯器第一次執行時，無法指定值給符號 BETA，因為這時候的 DELTA 尚未定義。結果第二次執行時也無法指定值給 ALPHA，原因出在於這個程式使用了 forward reference(DELTA)。解決的辦法就是多次執行的組譯器。

9.6. 資料結構的格式

9.6.1.靜態表格

一　機器操作碼表(MOT)

又稱為 OPTAB(Operation Code Table)，儲存符號式運算碼以及相對應的機器碼

- Pass1 時利用 OPTAB 來查詢與驗證原始程式的運算碼
- Pass2 時利用 OPTAB 將運算碼轉換成目的碼

二　虛擬操作碼(POT)

儲存虛擬指令的資訊

- Pass1 利用 POT 來處理原始程式的虛擬指令
- Pass2 利用 POT 來處理 Pass1 時無法處理的虛擬指令

9.6.2.動態表格

一　符號表(Symbol Table-ST)

儲存符號的名稱及位址職等資訊

- Pass1 將符號的定義存入 ST，且必須包含位址
- Pass2 搜尋 ST 以取得 symbol 的位址以產生目的碼

二　文字表(Literal Table-LT)

儲存文字的名稱，值，長度，及位址

- Pass1 遇到 literal 時候查詢 LT，若找到則不做任何的處理，若是找不到則將該文字加入 LT，但其位址尚未決定。當遇到 LTORG 或是 END 時候將決定此階段所有文字的位址
- Pass2 遇到 literal 查 LT 以產生目的碼

三　基底暫存表(Base register Table-BT)

儲存基底暫存器的名稱與其值

- Pass1 不會對 BT 有任何的處理動作
- Pass2 將搜尋 BT 以確定哪一個暫存器為基底暫存器及其值

9.7. 程式重定位

9.7.1.程式重定位的定義

將程式載入不同於原來載入的位址，就是重定位

9.7.2.絕對程式

程式必須載入到使用者指定的位址，才能執行的程式，就是絕對程式

9.7.3.可重定址程式

目的程式中，包含了在載入記憶體時需要修改的資訊，這類程式就稱之為可重定址程式。換句話說，目的程式會提供訊息給載入程式，由載入程式來做修正位址的動作。

9.7.4.修飾紀錄(Modification record)

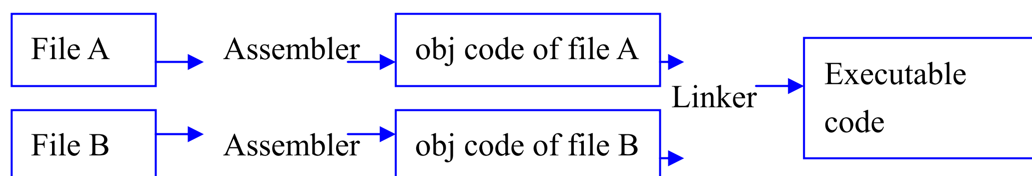
修飾紀錄記載了目的程式在載入記憶體時，需要修改的資訊。修飾紀錄的內容有以下兩種

- 被修改的位址欄的起始位址(相對於程式的起始位址)
- 被修改的欄位的長度

9.8. 程式區段

9.9. 控制段

9.9.1.控制段的意義



- 程式可以分成多個控制段，每個控制段可以個別組譯或編譯
- 所有的控制段會經由 Linker 連結在一起
- 允許外部參考

9.9.2.外部參考

當控制段必須參考不在自己的程式段內定義的指令或資料，這樣的情形就稱為外部參考

9.9.3.指令說明

♦ 設定控制段指令

CSECT

♦ 設定可供其他控制段參考的符號指令

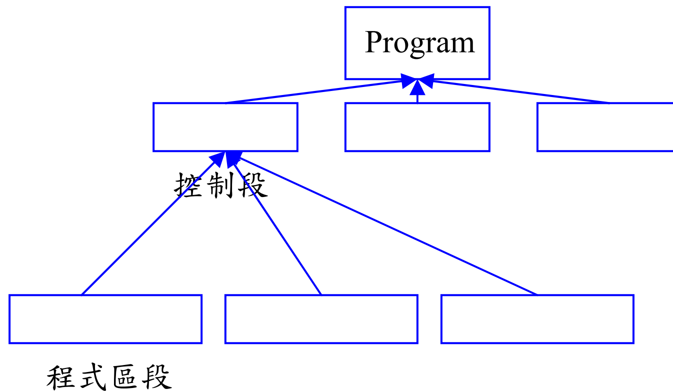
EXTDEF

♦ 設定使用外部符號的指令

EXTREF

9.9.4. 重要觀念

- 控制區段式程式的一部分，每一個控制段都可以獨立的載入記憶體中，也可以重新定位
- EXTDEF 指令：產生 Define record
- EXTREF 指令：產生 Reference record



範例4.
若提供了控制段的功能，則目的碼(object code)中應該增加哪些資訊？

Ans:

♦ **定義紀錄(Define record)**

紀錄所有在控制段中定義的外部符號，以及外部符號在控制端中的相對位址

♦ **參考紀錄(Reference record)**

紀錄所有在控制段中參考的外部符號名稱

♦ **修飾紀錄(Modification record)**

必須包含下列資訊

- 要修改的欄位的位址，相對於所屬的控制段的起始位址
- 欄位長度
- 修改旗標(+或-)
- 要加上(或減)的外部符號

9.10. 具有 Overlay 結構的兩次處理組譯程式

9.11. 常見的定址模式

9.11.1. 立即定址模式

運算元就是指令的一部分，執行時不用再做記憶體的存取動作。

9.11.2. 直接定址模式

又稱絕對定址。指令中的運算元為位址，必須透過位址去記憶體拿資料。

9.11.3. 間接定址模式

運算元中放的是位址，從位址拿到的資料又是位址，最後作一次記憶體存取才是資料。

9.11.4. 索引定址模式

將運算元的值加上索引暫存器的值就是資料的位址

9.11.5. 基底定址模式

將運算元的值加上基底暫存器的值就是資料的位址

9.11.6. 相對定址模式

將運算元的值加上程式計數器的值就是資料的位址

9.12. 考古題

範例5.

於組合語言中使用常數值(Literals)和(Immediate Addressing)有何不同？Assembler 如何處理上述情況？

何謂定址常數(Address Constant)？Assembler 如何處理位址常數？

Assembler 要提供哪些資訊給 Loader 做 Program Relocation 和 Linker 做程式連結？

Ans:

- **Literals**

組譯器會在其他記憶體位址產生指定的常數值，產生常數的位址會被當作是機器指令的目標位址使用。使用 Literal 的作用，就跟在其他地方先使用標籤定義常數，然後以該常數作為指令運算元是一樣的。

- **Immediate Addressing**

運算元的值會被組譯成為機器指令的一部分

Assembler 必須提供 H, T, E, M record，來作為 relocation 之用

- **H-record(標頭紀錄)**

即 Head record。紀錄程式的名稱，程式的起始位址以及程式的長度等三種資訊。

- **T-record(文本紀錄)**

即 Text record。紀錄程式的內容，包含目的碼起始的位址，指令的機器碼，欲載入的位址及資料

- **E-record(結束紀錄)**

即 End record。紀錄程式的結束處。並可以指定程式第一個開始執行的指令的位址。

- **M-record(修飾紀錄)**

修飾紀錄記載了目的程式在載入記憶體時，需要修改的資訊。修飾紀錄的內容有以下兩種

- 被修改的位址欄的起始位址(相對於程式的起始位址)
- 被修改的欄位的長度

而 D, R record 是用來做連結之用

第10章. 鏈結載入程式(linking loader)

10.1. 載入程式的定義與功能

10.1.1. 載入程式的定義

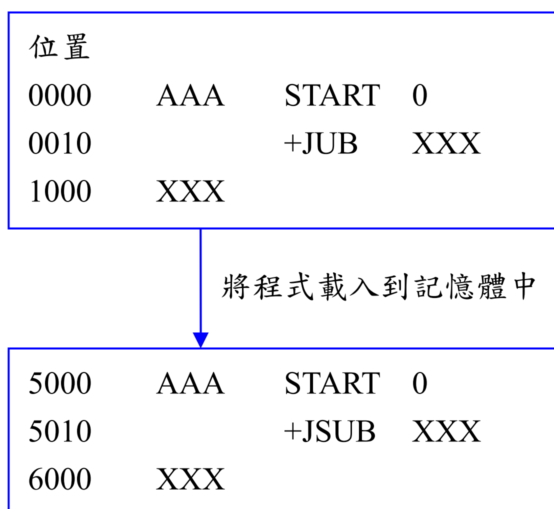
載入程式是將目的程式(object program) 處理過後載入(loading)到記憶體中

10.1.2. 載入程式的功能

- 配置記憶體空間(Allocation)：要求 OS 配置一塊足夠大的空間以供程式執行之用
- 連結(Linking)：解決不同程式段彼此相互參考的問題
- 重定址(Relocation)：載入程式可調整目的程式中與記憶體相關的指令或資料，使得程式能載入不同於原先載入的位子
- 載入(Loading)：將程式載入到記憶體，以供未來使用

10.2. 重新定址(relocation)

10.2.1. 基本觀念



10.2.2. 重定址載入程式

允許重定址的載入程式即為重定址載入程式。重定址載入程式又稱為相對載入程式

10.2.3. 重定址製作法

一　利用修飾紀錄

修飾紀錄含有

- 必須修改的欄位的起始位元
- 修改的長度
- 修改的方式。利用"+"或"-"符號來表示修改的方式

♦ **例子**

Ex. M^___^___^+AAA

♦ **優點**

對於使用相對定址的機器會比較適用，否則使用直接定址的機器，除了 RSUB 的指令幾乎都要修改，會需要很多修改紀錄，目的檔的大小會因此而暴增。

二 利用重定址位元

每一個 text record 中共含有 12 個指令，故利用 12 個重定位址分別對應 12 個指令，當重定位址位元為 1 的時候，代表必須重新定址，否則不必重新定址。

範例6. 若利用重定位位元來達到重定位的目的，請問文本記錄需要如何修正？

Ans:

♦ **Text record 原來的格式**

T 位址 長度 object code

♦ **Text record 新的格式**

T 位址 長度 重定位位元 object code

♦ **優點**

適用於直接定址並有固定指令格式的機器，如果將相對於目的碼的字組的重新定位位元設定為 1，程式載入時會將程式的起始位址加入到這個字組

10.3. 連結載入程式(Linking Loader)的製作

10.3.1. 爲什麼要做連結

一個程式可能由多個控制段所組成，這些個控制段可能彼此之間會有符號的參考問題，即外部參考。因此需要經過連結來處理，以解決不同程式區段間外部參考的問題。

10.3.2. 實例說明

♦ **PROGA 的程式段**

EXTDET A1,A2

EXTREF B1,B2,C1,C2

♦ **PROGB 的程式段**

EXTDEF B1,B2

EXTREF A1,A2,C1,C2

♦ **PROGC 的程式段**

EXTDEF C1,C2

EXTREF A1,A2,B1,B2

♦ **若在 PROGA 中有以下敘述**

PREF1 LDA B1 => 產生一個 M record

- Ex. M^__^__^+B1

REF2 word B2-B1+C1 => 產生三個 M record

- Ex. M^__^__^+B2
- M^__^__^-B1
- M^__^__^+C2

10.3.3. 製作法

一 資料結構

- ♦ **外部符號表(External Symbol Table)**

含有所有外部符號的資料

如上例說明的程式段，將建立下面的 ESTAB

Control Section	**Symbol name**	**Address**	**Length**
PROGA	A1 A2		
PROGB	B1 B2		
PROGC	C1 C2		

二 演算法

- Pass1:建立 ESTAB
- Pass2:Loading, Linking 與 Relocation

10.3.4. 多個修飾紀錄的問題

在前面介紹的方法，對於一個外部符號，會有多個修飾紀錄(Modification record)，因此查詢時必須以符號的名稱作為查詢的依據，在必須多次查詢下，這種作法的效率較差，因此可以利用以下方法

- ♦ **在 Refer record 中加入編號**
- ♦ **在 Modification record 中，以參考編號取代外部編號**
- ♦ **查詢法**

將各個外部符號的位址查出，放入一個一維陣列中，對於外部符號的查詢直接以參考編號做為陣列索引。

10.4. 一般載入程式

10.4.1. 一般載入程式的作法

- 程式經由 Translator 處理後產生目的碼(object code)
- 目的碼儲存於輔助記憶體中
- 要執行時由一般載入程式將目的碼載入到主記憶體

10.4.2. 一般載入程式的優點

- 較節省記憶體空間
- 較節省執行時間

10.5. Assembler and Go Loader

10.5.1. Assembler and Go Loader 的作法

這種 loader 包含了 assembler，先對 source program 做完組譯之後，直接將控制權轉移到程式的起始位址，開始程式的執行動作

10.5.2. 優點

- 容易製作

10.5.3. 缺點

- 較浪費記憶體空間
- 較浪費時間
- 程式設計缺乏彈性

10.6. 絕對載入程式

10.6.1. 絕對載入程式的功能

- Allocation: 程式設計師
- Linking: 程式設計師
- Relocation: 組譯程式
- Loading: loader

10.6.2. 優點

- 使用者有較大的彈性

10.6.3. 缺點

- 程式設計師必須安排 object code 的載入位址
- 程式設計師必須牢記副程式的位址
- Loader 無法處理重定位的問題，也就是說程式無法在不同的位址執行

10.7. 直接連結載入程式

直接連結載入程式為重定址載入程式的一種

10.7.1. 直接連結載入程式的功能

- Allocation: OS
- Linking:允許存在多個控制段，而且 Linking 工作由 loader 負責
- Relocation: loader
- Loading: loader

10.7.2. 目的碼的資訊

由於直接連結載入程式必須負責 linking、Relocation 以及 loading 的工作，因

此目的碼中必須包含以下的資訊

- 程式的長度
- 在本程式段中定義，但可被其他程式段引用的符號名稱
- 在本程式段中引用，但不在本程式段中定義的符號名稱
- 在本程式段中所有與位址有關的資料
- 原始程式被組譯程式處理過後產生的目的碼
- 程式開始執行的位址以及程式結束處

範例7. 以上說明的這些資訊，通常會包含在哪些資料結構中？

Ans:

- H-Record
- T
- E
- D
- R
- M-Record(所有與位址有關的資料)

10.7.3. DLL 的兩次處理

DLL 一般是採兩次處理方式製作，動作如下

- ♦ **Pass 1**

解決連結問題，即處理所有外部符號參考(external symbol reference)的問題

- ♦ **Pass 2**

解決重定位(Relocation)及載入(Loading)的問題

10.7.4. DLL 的優點

- 程式可分成多個控制段
- 各個控制段可彼此互相參考其他控制段內的資料
- 控制段可個別組譯

10.7.5. DLL 的缺點

- DLL 佔用相當大的記憶體空間
- DLL 處理的動作較費時
- 每次執行都必須重新做"Linking"的動作

10.8. BSS Loader (Binary Symbolic Subroutine Loader)

BSS 屬於可重定址載入程式

10.8.1. BSS 的功能

- Allocation: 根據組譯程式提供的程式長度來做配置
- Linking:利用轉移向量，來完成 linking
- Loading: loader

10.8.2. BSS loader 特性介紹

- 允許有多的程式段，但只有允許一個資料段
- 利用轉移向量處理 linking 問題

10.8.3. BSS 缺點

- 轉移向量會浪費額外的記憶體空間
- 只允許一個資料段
- 利用副程式呼叫，所以執行的速度較慢

10.9. 連結編輯程式

10.9.1. 連結編輯程式的作法

先對不同的控制段作 Linking 的的動作，產生出「Linked Program」(即 load module)。待執行程式時才執行重定址(relocation)及載入(loading)動作

10.9.2. 連結編輯程式的優點

- 具有較高的執行效率
- 執行時不需處理 Linking 的動作

10.10. 自動程式庫搜尋

10.11. 載入程式的功能選擇

10.12. 覆疊程式(Overlay Program)

10.13. 動態連結

10.13.1. 基本觀念

在程式的執行過程中，即有可能程式中有許多的副程式並不會被執行到，因此假如在連結時候也將這些副程式連結進來，不僅增加 linker 的負擔，也增加了記憶體的使用量。

10.13.2. 動態連結的意義

將程式的連結與載入的動作，延遲到程式執行的時候才處理；也就是真正會被執行到的程式，才做連結與載入的動作。動態連結又稱為「load on call」

10.13.3. 動態連結的製作

程式對 OS 提出一個系統呼叫，並且將副程式的名稱傳給 OS，然後將程式的控制權轉移給 OS

作業系統檢查此副程式是否已經在記憶體中，有以下兩種狀況

- ♦ **已經在記憶體中**
  - 直接呼叫副程式
  - 副程式執行工作
  - 副程式執行完畢把控制權交還給 OS
  - OS 將控制權交還給程式
- ♦ **不在記憶體中**
  - 由動態連結程式庫(Dynamic Link Library)終將副程式連結並載入記憶體中
  - OS 呼叫新載入的副程式
  - 副程式執行
  - 副程式執行完畢，並將控制權交還給 OS
  - OS 將控制權交還給程式

10.13.4. DLL 的優缺點

- ♦ **優點**
  - 較節省記憶體空間
  - 較節省連結的時間
  - 程式的結構可傳動態性的修改
- ♦ **缺點**
  - 執行時間較長，並增加執行程式的負擔

10.14. 考古題

範例8.
何謂 binding？舉例說明五種不同 binding 的時間

Ans:

♦ **Binding 的意義**

Binding 是將變數的名稱與其位址或值或屬性結合的動作，或處理不同程式段間彼此互相參考的問題，並解決 relocation 的問題。

Binding 時間	**Loader**
在組合(編譯時)	Absolute
在組合後，載入前	Linking editor
在載入時	BSS loader
在執行時動態 bind	Dynamic binder
在執行時，真正參考到時	Dynamic linking loader(DLL)

第11章. 巨集處理程式

11.1. 基本觀念

寫程式常會把一些相關或類似的敘述集合在一起，以節省程式設計的時間，常用的方法有以下三種

- 迴圈
- 副程式
- 巨集

11.2. 巨集的基本概念介紹

11.2.1. 巨集

又稱為巨集指令，代表程式中一群常用的敘述

11.2.2. 巨集定義

定義巨集和其對應的一群敘述

11.2.3. 巨集展開

巨集展開又稱為巨集呼叫(macro call)，將巨集名稱以相對應的一群敘述取代

```
Example   Macro   &ARG
          L       1,&ARG
          A       1,=F'1'
          ST      1,&ARG
          Mend
```

- Example 表示巨集名稱
- 中間三行敘述以 Example 代表其名稱，也就是必須透過 Example 來呼叫巨集

11.3. 巨集與副程式的區別

一　就記憶體空間而言

副程式較節省記憶體空間

二　就執行時間而言

巨集較節省執行時間

三　就參數而言

副程式：值或位址取代

巨集：字串取代

11.4. 巨集中參數處理的方式

11.4.1. 巨集指令引數

為了解決巨集只能將整個段落原封不動的代換巨集召用這種缺乏彈性的缺

點，因此我們在巨集指令中加上引數

- 在巨集定義中加上虛擬引數（Dummy Argument），
- 在巨集召用中則加入相對應的召用參數（Calling Parameter），

如此，即可使每次擴展的內容有相當彈性的改變。

11.4.2. 巨集參數處理的方式有以下兩種

一　位子參數

♦ 作法

依據實際參數(actual parameter)與形式參數(formal parameter)的順序作一對一的對應

♦ 實例

Ex1 Macro &Var1, &Var2, &Var3

- 呼叫方法：Ex1 A, B, C, D

二　關鍵字參數

♦ 作法

每個形式參數之後直接指定對應的實際參數名稱

♦ 實例

Ex1 Macro &Var1=A, &Var2, &Var3=C, &Var4

- 呼叫方法：Ex1 &VAR2=B, &Var4=D

11.5. 巨集定義與巨集呼叫間的關係

11.5.1. 在巨集中呼叫巨集

一　意義

在巨集定義中，呼叫巨集

二　實例

假設有下列兩個 macro definitions

```
MACRO                  MACRO
ADD1      &ARG         ADDS &ARG1,&ARG2,&ARG3
L         1,&ARG       ADD1 &ARG1
A         1,=F'1'      ADD1 &ARG2
ST        1,&ARG       ADD1 &ARG3
MEND                   MEND
```

試 expand 下列的 macro instruction

ADD DATA1,DATA2,DATA3

Ans:

```
L    1,DATA1
A    1,=F'1'
ST   1,DATA1
```

```
L    1,DATA2
A    1,=F'1'
ST   1,DATA2
L    1,DATA3
A    1,=F'1'
ST   1,DATA3
```

11.5.2. 巨集定義內含巨集定義

一 意義

二 實例

```
Ex2     Macro   &A      &B
&A      Macro   &C
        LDA     &C
        &B      #1
        STA     &C
        Mend
        Mend
```

♦ 巨集呼叫

```
Ex2         SUB1,   tmp1
=>SUB1      Macro   &C
            LDA     &C
            tmp1    #1
            STA     &C
            Mend

Ex2         SUB2,   tmp2
=>SUB2      Macro   &C
            LDA     &C
            tmp2    #1
            STA     &C
            Mend
```

經過這兩個巨集展開的動作後，以後就可以直接呼叫 SUB1 與 SUB2 這兩個巨集

11.6. 巨集中標記(Label)的處理方式

11.6.1. Label 重複出現的基本觀念

若某個巨集中存在 label，當巨集被呼叫兩次以上時，將使得 label 重複出現兩次以上，也就是發生符號多重定義的錯誤。

11.6.2. 解決辦法

一　計數法

♦　作法

將各個巨集展開的次數利用一個計數器來記錄，並將此值附加在 label 之後，如此就可以產生唯一的 label 名稱

二　參數式標記法

♦　作法

將巨集中的標記名稱視為參數，每次呼叫巨集，都必須指定 label 的名稱

三　區域標記法

♦　作法

將標記宣告為區域性(local)label，如此一來巨集處理程式將自行處理標記可能發生的問題，使用者就不用處理這些問題了

11.7. 條件式巨集展開

11.8. 巨集處理程式的製作

11.8.1. 基本的資料結構

一　DEFTAB(巨集定義表)

DEFTAB 中會存放巨集定義的程式段

二　NAMTAB(巨集名稱表)

NAMTAB 的內容包含了三個部份，分別為

- 巨集名稱
- 巨集名稱相對應的起始位子(index)
- 巨集名稱相對應的結束位子(index)

三　ARGTAB(參數表)

ARGTAB 中會存放參數名稱

11.8.2. 範例

11.9. 巨集組譯程式

一　巨集組譯程式的基本觀念

將巨集處理程式和組譯程式結合在一起

♦　作法

將巨集處理程式的工作併入組譯程式的 pass1 來處理。將工作分配如下

Pass1:

- A Macro definition(存入 DEFTAB 中)
- B Macro expanding(由 DEFTAB 中輸入)
- C Define symbol

Pass2:

- Code generation

二 優點

- 減少一個 pass 的處理
- 資料結構可共用
- 副程式可共用
- 錯誤訊息可一併列出

三 缺點

- 較耗費記憶體空間
- 較難設計

第12章. 語法與編譯程式

12.1. 編譯程式基本概念

12.2. P-碼編譯器(P-code compiler)

12.3. Compiler's compiler

12.4. 交互式編譯程式

12.5. 自動機(Automata)理論

12.5.1. 自動機

自動機可以讀取一個字串，來判斷這個字串是否屬於此語言

12.5.2. 有限自動機(Finite Automata-FA)

一 特性

- 沒有記憶體
- Finite state system
- 當某個輸入產生時，系統會由一個狀態轉換到另外一個狀態

二 種類

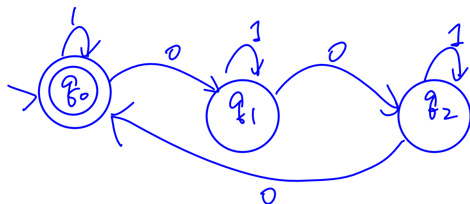
- Deterministic Finite Automata(DFA)
- Nondeterministic Finite Automata(NFA)

12.5.3. DFA(Deterministic Finite Automata)

$L=\{w \mid w \in \{0,1\}^*$, w中0的數目為3的倍數$\}$

Find a DFA to accept L

Ans:



12.5.4. NFA(Nondeterministic Finite Automata)

一 NFA 定義

- 只要有一條路徑可以到 final state，就可以接受此字串
- 對同一個 input 可能可以進入數個不同的 state

12.6. 語法的基本定義

12.7. 文法的要素(組成要素)

- N:非終端符號
- T:終端符號
- S:起始符號
- P:文法產生規則

12.8. 文法的分類(Classification of grammers)

2.文法的分類：

12.8.1. Type 0：無任何限制。

Turing machine

12.8.2. Type 1：與上下文相關的文法

與上下文相關的文法 (context-sensitive grammar)其文法產生規則必須滿足：

$\forall u \rightarrow v \in P$， $|u| \leq |v|$

U 的長度必須小於或等於 v 的長度

Linear bounded automata 採用

12.8.3. Type 2：與上下文無關的文法 (context-free grammar)

與上下文無關的文法 (context-free grammar)其文法產生規則必須滿足

$\forall u \rightarrow v \in (N \cup T)^*$

其中 $u \in N$ ，$v \in (N \cup T)^* - \lambda$

註：Type 2 grammar 即為 B.N.F. 文法。

Pushdown automata 採用，又稱 parser

12.8.4. Type 3：正規文法 (regular grammar)

正規文法又分為二類：

- ♦ **右線性正規文法 (right linear regular grammar)**

其文法產生規則需滿足：

$A \rightarrow uB$ or $A \rightarrow u$

其中 $A,B \in N$，$u \in T$

- ♦ **左線性正規文法 (left linear regular grammar)**

其文法產生規則則必需滿足：

$A \rightarrow Bu$ or $A \rightarrow u$

其中 $A,B \in N$，$u \in T$

Finite State machine 採用，又稱作 lexical 或是 scanner

12.9. Backus Naur Form(BNF 文法)

(1)B.N.F. 文法即 type 2 grammar (context-free grammar)。

(2)B.N.F.文法符號說明:

a. “::=” :表示“定義為”。

B. “{}” :表示出現 0 次,1 次,...。

c. “〔 〕” :表示出現 0 次或 1 次。

d. “ | ” :表示 OR。

12.10. 剖析樹(parse tree)

12.11. 模擬兩可的文法(ambiguous grammar)

12.12. 描述程式語言語法的方式

12.13. 剖析法(Parsing)

12.13.1. Top down

一 Recursive Descent Parser

◆ **原理**

Recursive Descent Parser 利用 Recursive Procedure 來辨認輸入字串，且不需做 backtracking 的動作，因此不允許有立即左遞迴及左因子的情況發生。

<exp>::=<term>|<exp>+<term>|<exp>-<term>

兩種轉換方式

① <exp>::=<term>{+<term>|-<term>}

② <exp>::=<term><exp'>

<exp'>::=+<term><exp'>|-<term><exp'>|λ

12.13.2. Bottom up

一 Shift Reduce Parser

◆ **目標**

將一個句子根據文法產生規則，簡化成起始符號

◆ **演算法**

利用一個 stack 做為分析的工具，分析方法如下

- 將輸入字串的 token 依照順序移入 stack 中
- 每次加入 token 進入 stack 後，都需要比較 stack 頂端的符號是否符合某一條文法規則，若符合則以該規則左端的符號來取代 stack 頂端的符號，若不符合則繼續將輸入字串的下一個 token 移入 stack
- 重複執行上面兩個動作，直到 stack 中只有起始符號為止

二　演算子的優先順序(Operator Precedence Parser)

♦　**基本觀念**

根據一個事先建立的運算子順位來 parse 一個句子

♦　**建立運算子優先順序表**

♦　**演算子的優先順序的演算法**

比較 stack 頂端的 terminal symbol A 與 Input string 的第一個 token B

If A<B then push “<” and B into stack

Else if A=B then push B into stack

Else “Reduce”

12.14.　最佳化(Optimization)

12.14.1.　與機器相關的最佳化

- 刪除多餘的 STORE 與 LOAD 指令
- 多利用暫存器來計算
- 利用效率較高的指令來取代效率較低的

12.14.2.　與機器無關的最佳化

- 刪去共同的副運算式
- 編譯時期的計算
- 將迴圈的不變量移到迴圈外
- 捷徑計算

記憶法：回覆便捷