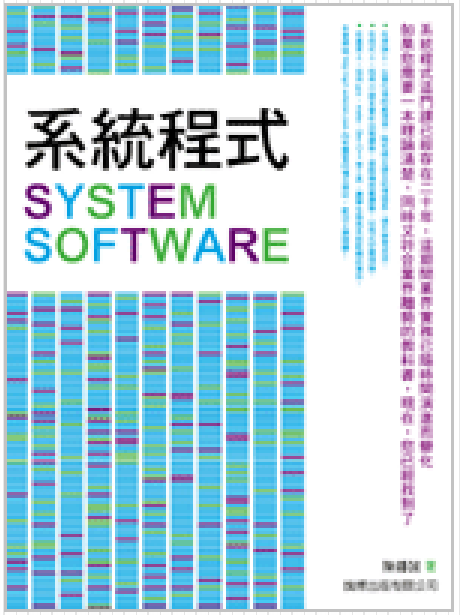
第 8 章、編譯器

作者：陳鍾誠

旗標出版社



第 8 章、編譯器

- 8.1 簡介
- 8.2 詞彙掃描
- 8.3 語法剖析
- 8.4 語意分析
- 8.5 中間碼產生
- 8.6 組合語言產生
- 8.7 最佳化
- 8.8 實務案例：gcc 編譯器

8.1 簡介

- 編譯器
  - 將高階語言轉換成組合語言(或機器碼) 的工具

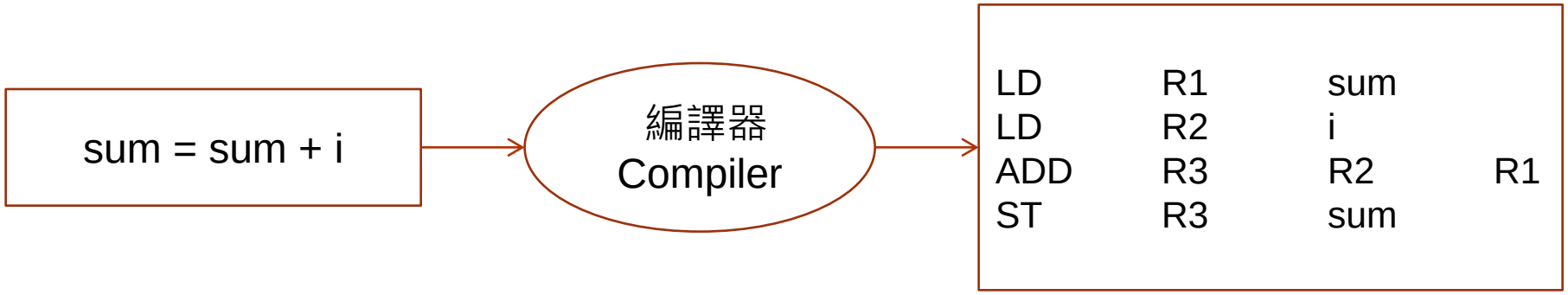


圖 8.1 編譯器的輸入與輸出

C0 語言

- C0 語言
  - 代表 C 語言第 0 版的意思，就是小型的 C 語言。
  - 只包含 for 迴圈與基本運算。

- 範例

範例 8.1 一個 C0 語言的程式範例

```
C0 語言程式 (位於 sum.c0 範例檔中)
sum = 0;
for (i=1; i<=10; i++)
{
  sum = sum + i;
}
return sum;
```

C0 語言的 EBNF 語法

- 星號 * 代表重複零次以上
- 加號 + 代表重複一次以上
- 問號 ? 代表重複 0 或 1 次

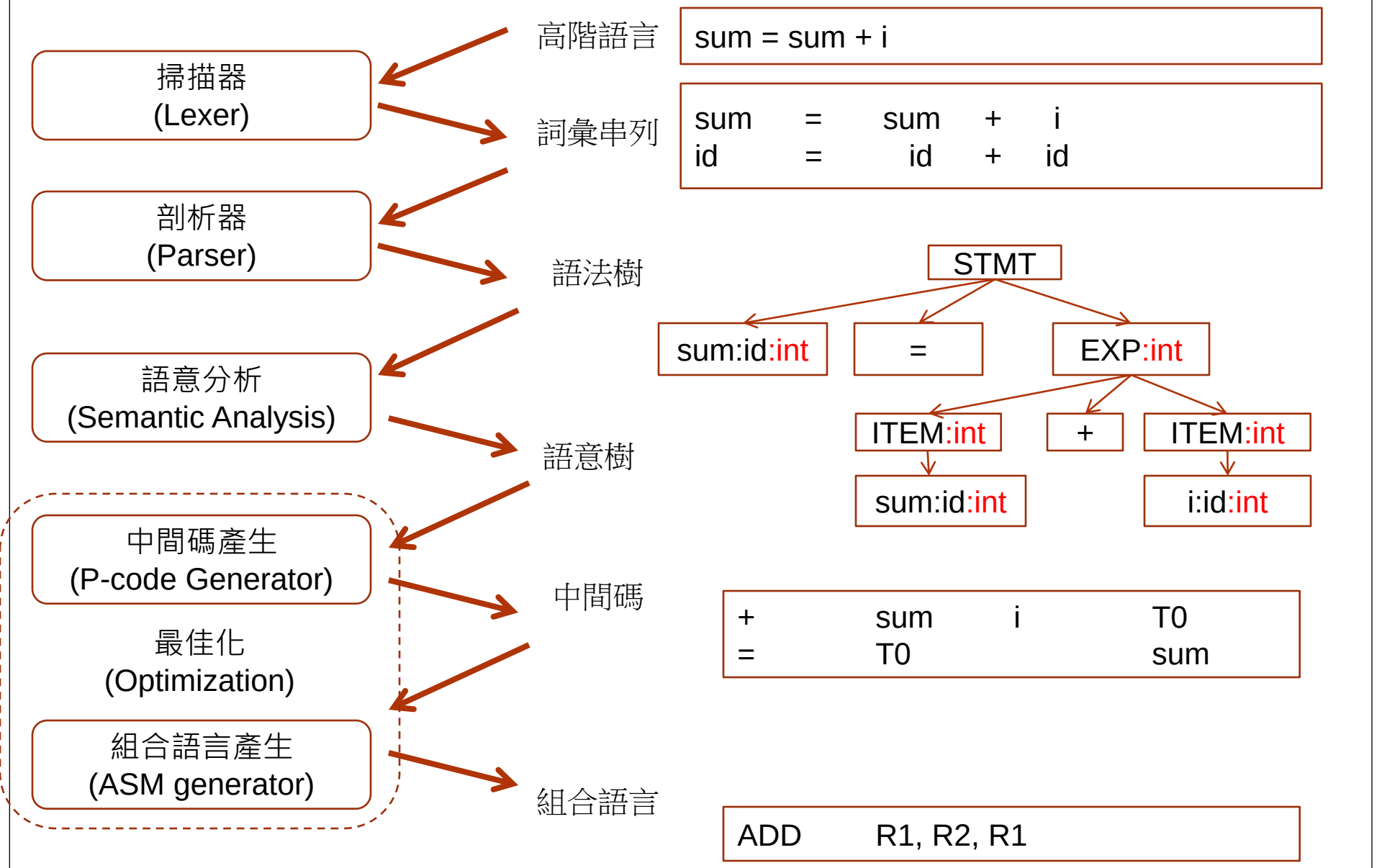
▶圖 8.2 C0 語言的 EBNF 語法規則

編號	EBNF 語法規則
1	PROG = BaseList
2a	BaseList = (BASE)*
3	BASE = FOR | STMT ';'
4	FOR = 'for' '(' STMT ';' COND ';' STMT ')' BLOCK
5	STMT = 'return' id | id '=' EXP | id ('++'|'--')
6	BLOCK = '{' BaseList '}'
7a	EXP = ITEM ([+-*/] ITEM)?
8	COND = EXP ('=='|'!='|'<='|'>='|'<'|'>') EXP
9	ITEM = id | number
10	id = [A-Za-z_][A-Za-z0-9_]*
11	number = [0-9]+

編譯器的六大階段

- 1. 詞彙掃描 (Scan 或 Lexical Analysis)
  - 將整個程式分成一個一個的基本詞彙 (token)
- 2. 語法剖析 (Parsing 或 Syntax Analysis)
  - 剖析器利用語法規則進行比對, 以逐步建立語法樹。
- 3. 語意分析 (Semantic Analysis)
  - 為語法樹加註節點型態, 並檢查型態是否相容, 然後輸出語意樹
- 4. 中間碼產生 (Pcode Generator)
  - 語意樹被轉換成中間碼
- 5. 最佳化 (Optimization)
  - 考慮暫存器的配置問題, 降低指令數量，增加效率。
- 6. 組合語言產生 (Assembly Code Generator)
  - 將中間碼轉換為組合語言輸出。

圖 8.3 編譯器的六大階段



8.2 詞彙掃描

- 詞彙掃描
  - 將程式切分成一個一個的詞彙 (token)

- 範例 1：sum = sum + i

圖 8.4 掃描階段的輸出–具類型標記的詞彙串列

詞彙 (Token)	sum	=	sum	+	i
詞類標記 (Type)	id	=	id	+	id

- 範例 2：printf("%d", 30)

圖 8.5 掃描階段的輸出–包含常數與字串的範例

詞彙 (Token)	printf	(	"%d"	,	30	)
詞類標記 (Type)	id	(	string	,	number	)

詞彙掃描的程式

▶圖 8.7 將 C0 語言的程式分解成詞彙的演算法

C0 語言的掃描器演算法

```
Algorithm tokenize(file)
   c = file.nextchar()
   while not file.isEnd()
      token = nextToken(file, c)
      print(token)
   end while
End Algorithm
```

說明

```
file 為原始程式碼檔案
   取得第一個字元
   在檔案尚未結束時
      取得下一個詞彙
      輸出該詞彙
```

以正規表達式進行掃描

- 整數
  - number = [0-9]+
- 名稱
  - id = [A-Za-z_][A-Za-z0-9_]*

使用迴圈逐字進行掃描

	C0 語言的掃描器演算法	說明
1	`function nextToken(file, c)`[2]	file 為原始程式碼檔案, c 為下一個字元
2	`  token = new string()`	建立 token 字串
3	`  if (c in [0-9])`	如果是數字 (number)
4	`    while (c in [0-9])`	不斷取得數字
5	`      token.append(c)`	放入 token 字串中
6	`      c = file.nextchar()`	再取得下一個字元
7	`    end while`	
8	`    tag = "number"`	設定詞類標記(tag)為數字
9	`  else if (c in [a-zA-Z_])`	如果是英文字母 (id)
10	`    while (c in [a-zA-Z0-9_])`	不斷取得英文、數字或底線
11	`      token.append(c)`	放入 token 字串中
12	`      c = file.nextchar()`	再取得下一個字元
13	`    end while`	
14	`    if token is keyword`	如果是關鍵字(C0 的關鍵字只有 for)
15	`      tag = token`	設定詞類標記為該關鍵字
16	`    else`	否則
17	`      tag = id`	設定詞類標記為 id
18	`    end if`	
19	`  else if (c in [+-*<=>!])`	如果是運算符號
20	`     while (c in [+-*<=>!])`	不斷取得運算符號
21	`       token.append(c)`	放入 token 字串中
22	`       c = file.nextchar()`	再取得下一個字元
23	`     end while`	
24	`   tag = token`	設定詞類標記為該詞彙
25	`  else`	否則就是單一字元, 像是 { 或 }
26	`    token = c;`	設定 token 為該字元
27	`    tag = token;`	設定 tag 為該字元
28	`  end if`	
29	`  return (token, tag)`	傳回取得的詞彙
30	`end`	

8.3 語法剖析

- 剖析器
  - 由上而下的剖析法
    - 像是遞迴下降法、LL 法
  - 由下而上的剖析法
    - 像是運算子優先矩陣法、LR 法

- 本章所採用的方法
  - 遞迴下降法
    - 其餘方法請參考編譯器的專門書籍。

遞迴下降法

- 方法
  - 剖析器的撰寫者, 只要能夠將 EBNF 語法轉換為遞迴下降函數, 就能製作出遞迴下降剖析器。

- 轉換方法

規則：A = b B | c C

```
if isNext(b) {
  next(b);
  parseB();
} else if isNext(c) {
  next(c);
  parseC();
}
```

規則：A = (b B)*

```
while (isNext(b)) {
  next(b);
  parseB();
}
```

遞迴下降法 – 範例 1

▶圖 8.8 將規則 EXP= ITEM ([+-*/] ITEM)? 翻譯成遞迴下降剖析程式

C0 語言 EXP 規則的剖析函數

```
function parseExp()
   pushNode("EXP")
   parseItem();
   if isNext("+|-|*|/")
      next("+|-|*|/")
      parseItem()
   end if
   popNode("EXP")
end
```

說明

```
剖析 EXP 語法
    建立 EXP 節點, 推入堆疊中
    剖析 ITEM 語法
    如果下一個是加減乘除符號
        取得該符號
        剖析下一個 ITEM 語法

    取出並傳回 EXP 這棵語法樹
```

C0 語言的剖析器 (第1頁)

▶圖 8.9 C0 語言的剖析器演算法

C0 語言的掃描器演算法	說明
Algorithm C0Parser	剖析器演算法
Stack stack	共用變數，包含 stack ：堆疊
File file	file ：輸入的程式檔
Token token	token: 目前的詞彙
char c	c ：掃描器的目前字元
// functions for parser	函數區開始
function parse(fileName)	剖析器的主要函數 - parse()
stack = new stack()	宣告堆疊
file = new File(fileName)	取得輸入檔，建立物件
c = file.nextchar()	取得第一個字元
getNextToken()	取得第一個 (詞彙，標記)
parseProg()	開始剖析該輸入程式檔
end	
function getNextToken()	取得下一個 (詞彙，標記)
(token, tag) =	取得下一個 (詞彙，標記)
nextToken(file, c)	
end	

CO 語言的剖析器 (第2頁)

```
function isNext(tags)
if (tokenNext.tag in tags)
  return true;
else
  return false;
end
function next(tags)
  if isNext(tags)
   child = new Node(token);
   parent = stack.peek()
   parent.addChild(child)
  end if
end
function pushNode(tag)
  node = new Node(tag)
  stack.push(node)
end
function popNode(tag)
  node = stack.pop()
  if (node.tag == tag)
   parentNode = stack.peek()
   parentNode.addChild(node)
  else
   error("Parse error")
  end if
end
```

判斷下一個詞彙標記是否為 tags 之一

將下一個詞彙建立為新節點，放入父節點中

建立具有 tag 標記的新節點，推入堆疊中

取出節點
- 取出節點
- 看看是否具有 tag 標記
  - 取得上一層的樹
  - 將該節點設定為上一層的子節點
- 如果不具有 tag 標記
  - 則是語法錯誤，進行錯誤處理

co 語言的剖析器 (第3頁)

```
function parseProg()
  pushNode("PROG")
  parseBaseList()
  popNode("PROG")[3]
end
function parseBaseList()
  pushNode("BaseList")
  while not file.isEnd()
   parseNext("BASE")
  popNode("BaseList")
end
function parseBase()
  pushNode("BASE")
  if isNext("for")
   parseFor()
  else
   parseStmt()
   next(";");
  end if
  popNode("BASE")
end
```

```
剖析規則 1:
  PROG = BaseList

剖析規則 2a:
  BaseList = (BASE)*

剖析規則 3: BASE = FOR | STMT ';'

  處理 FOR

  處理 STMT ';'
```

CO 語言的剖析器 (第4頁)

```
function parseFor()
   pushNode("FOR")
   next("for")
   next("(")
   parseStmt()
   next(";")
   parseCond()
   next(";")
   parseStmt()
   next(")")
   parseBlock()
   popNode("FOR")
end
function parseStmt()
   pushNode("STMT")

   if (isNext(p, "return"))
    next(p, "return");
    next(p, "id");
   else
    next(id)
    if isNext("=")
       next("=")
       parseExp()
    else
       next("++|--")
    end if
   end if
   popNode("STMT")
end
```

```
剖析規則 4:
   FOR =
      'for'
      '('
      STMT
      ';'
      COND
      ';'
      STMT
      ')'
      BLOCK


剖析規則 5:
   STMT = 'return' id | id '=' EXP |
                  id ('++'|'--')
   處理 'return' id

   處理 id '=' EXP

   處理 id ('++'|'--')
```

CO 語言的剖析器 (第5頁)

```
function parseBlock()
  pushNode("BLOCK")
  next("{")
  parseBaseList()
  next("}")
  popNode("BLOCK")
end
function parseExp()
  pushNode("EXP")
  parseItem();
  if isNext("+|-|*|/")
   next("+|-|*|/")
   parseItem()
  end if
  popNode("EXP")
end
```

```
剖析規則 6:
  BLOCK      = '{' BaseList '}'

剖析規則 7a:  EXP = ITEM ([+-*/] ITEM)?
  建立 EXP 節點, 推入堆疊中
  剖析 ITEM 語法
  如果下一個是加減乘除符號
    取得該符號
    剖析下一個 ITEM 語法

  取出並傳回 EXP 這棵語法樹
```

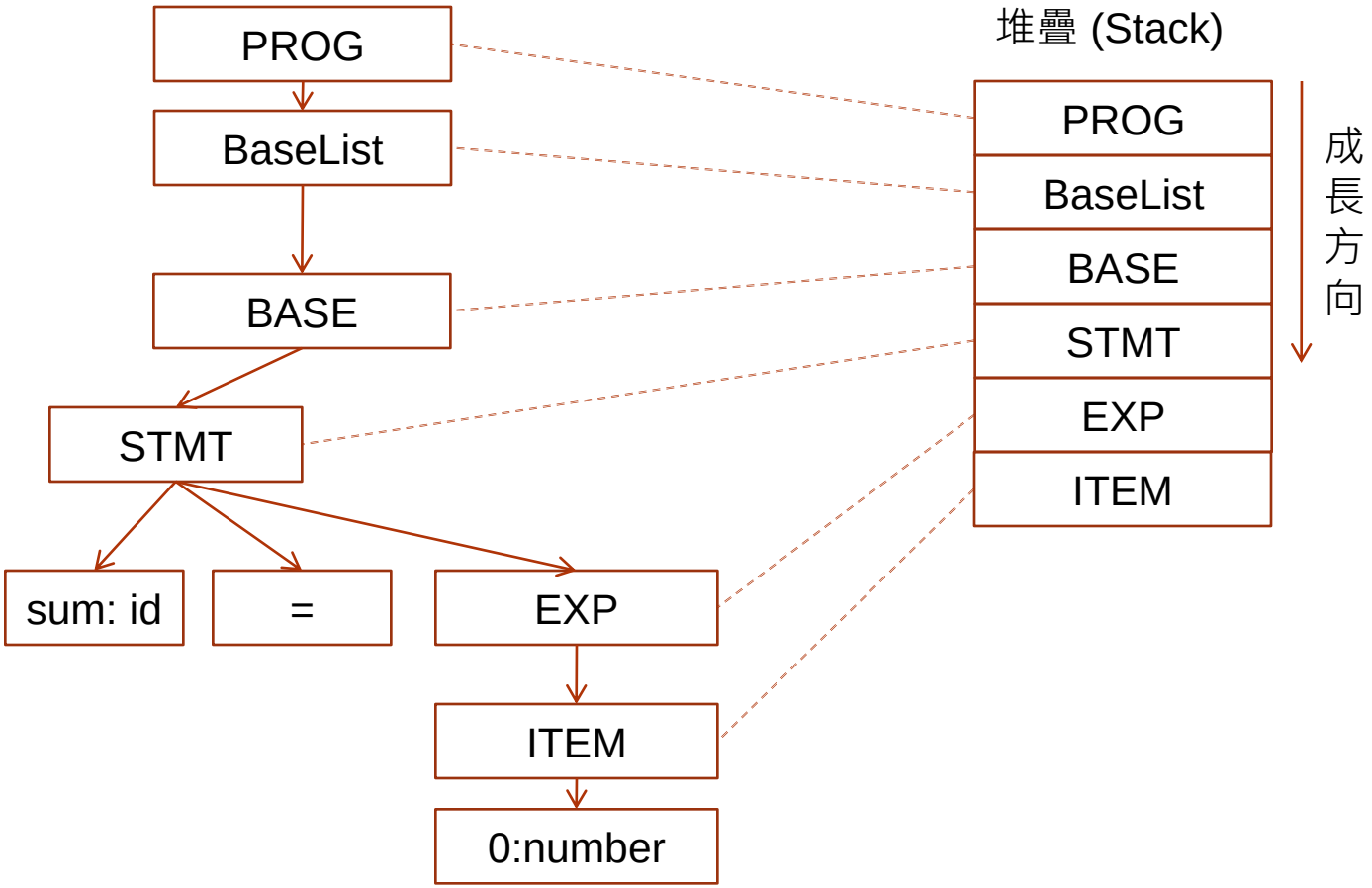
C0 語言的剖析器 (第6頁)

```
    function parseCond()
      pushNode("COND")
      parseExp()
      next("==|!=|<=|>=|<|>")
      parseExp()
      popNode("COND")
    end
    function parseItem()
      pushNode("ITEM")
      next("id|number");
      popNode("ITEM")
    end
end Algorithm
```

```
剖析規則 8: COND
  = EXP ('=='|'!='|'<='|'>='|'<'|'>')
    EXP
```

```
剖析規則 9:
  ITEM = id | number
```

圖 8.10 <範例 8.1> 剖析到 sum=0 時的語法樹與堆疊結構

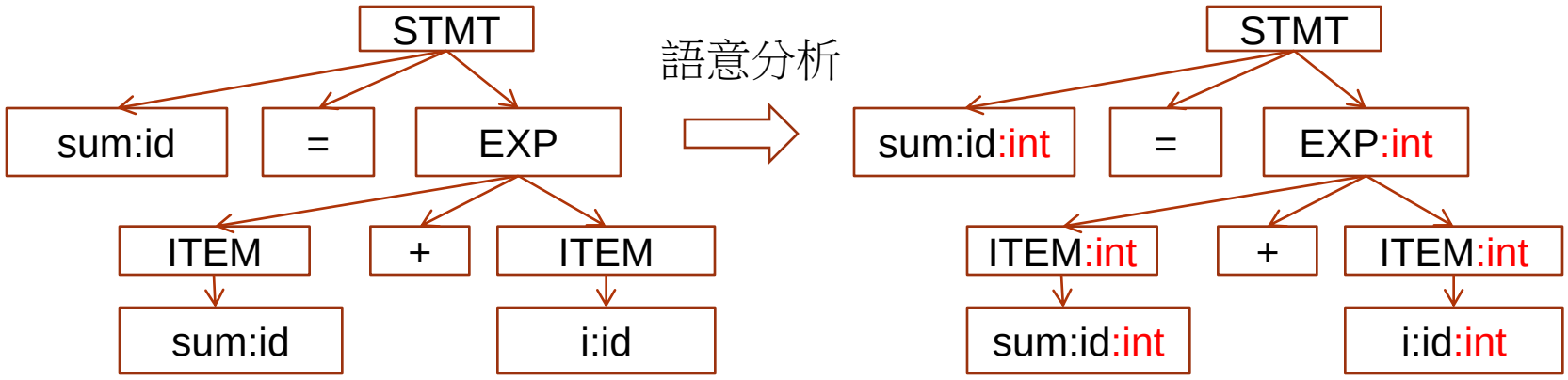


8.4 語意分析

- 執行時機
  - 當語法樹建立完成後, 緊接著通常會進行語意分析的動作。

- 執行動作
  - 語意分析必須確定每個節點的型態, 並且檢查這些型態是否可以相容, 然後才輸出具有標記的語法樹, 也就是語意樹。

語意分析的範例



(a) 語法樹

(b) 語意樹

圖 8.11 語意分析：將語法樹標註上節點型態

語法正確但語意錯誤的程式範例

- 範例 8.2
  - 語意分析器將會發現到 a 與 b 是無法相乘的, 因而輸出錯誤訊息。

▶範例 8.2 語法正確但語義錯誤的程式範例

C 語言程式	語義分析的結果

`int a=5, c;`
`char b[10];`
`c = a * b;`
	a, c 是整數
	b 是字元陣列
	c (錯誤?) = a (整數) * b (字元陣列)

8.5 中間碼產生

- 使用時機
  - 一旦語意樹建立完成, 我們就可以利用程式將樹中的每個節點, 展開為中間碼
- 方法
  - 利用遞迴的方式, 從根節點開始, 遞迴的展開每個子節點, 直到所有節點的中間碼都產生完畢為止。

將 C0 語言編譯成中間碼的範例

範例 8.3 將 C0 語言編譯成中間碼的範例

	C0 語言程式	中間碼	說明
1	sum = 0;	= 0 sum	設定 sum 為 0
2	for (i=1;i<=10;i++)	= 1 i	設定 i 為 1
3	{	FOR0:	for 迴圈的起始點
4	sum = sum + i;	CMP i 10	i >=< 10 ?
5	}	J > _FOR0	if (i>10) goto FOR1
6	return sum;	+ sum i T0	T0=sum + i
7		= T0 sum	sum = T0
8		+ i 1 i	i = i + 1
9		J FOR0	
10		_FOR0:	
11		RET sum	傳回 sum

中間碼產生的演算法

- 規則：STMT = id ‘=’ EXP
- 範例：sum = sum + i
  - 呼叫 generate(exp) ：其中的 exp 為 sum + i
  - pcode 一行會產生 = T0 sum
  - 並傳回 T0 作為 sum = sum + i 運算式的值

圖 8.12 產生 id=EXP 中間碼的演算法

演算法	說明

Algorithm generate(node)
…
else if (node.tag=STMT)
id = node.childs[0].token;
exp = node.childs[2];
expVar = generate(exp);
pcode("", "=", expVar, "", id);
return expVar;
…
End Algorithm
	處理 STMT 陳述
	取得 id
	取得 EXP 節點
	產生 EXP 的程式，並傳回變數 (例如 T0)
	產生指定敘述 (例如 =T0 sum)
	傳回臨時變數 (例如 T0)

圖 8.13 中間碼產生的演算法(第1頁)

中間碼產生的演算法	說明
Algorithm generate(node)	產生中間碼的演算法
if (node.tag=FOR)	處理 FOR 迴圈
stmt1 = node.childs[2]	語法：for (STMT;
cond = node.childs[4]	COND;
stmt2 = node.childs[6]	STMT)
block = node.childs[8]	BLOCK
generate(stmt1);	產生 STMT 的程式
tempForCount = forCount++;	取得下一個 for 標記代號
forBeginLabel = "+FOR" + tempForCount;	for 迴圈的起始標記(+FOR0)
forEndLabel = "-FOR"+tempForCount;	for 迴圈的結束標記(-FOR0)
pcode(forBeginLabel+":", "", "", "", "");	輸出迴圈起頭標記 (FOR0:)
condOp = generate(cond);	產生比較指令 (CMP i 10)
negateOp(condOp, negOp);	將運算反向 (i<=10 變 >)
pcode("", "J", negOp, "", forEndLabel);	輸出跳離指令 (J > _FOR0)
generate(block, nullVar);	產生 BLOCK 的程式
generate(stmt2, nullVar);	產生 STMT 的程式
pcode("", "J", "", "", forBeginLabel);	跳回到迴圈起頭 (J +FOR0)
pcode(forEndLabel, "", "", "", "");	輸出迴圈結束標記 (-FOR0:)
return NULL;	for 迴圈無傳回值

圖 8.13 中間碼產生的演算法(第2頁)

```
else if (node.tag=STMT)
   id = node.childs[0].token;
   if (node.childs[1].tag = "=")
    exp = node.childs[2];
    expVar = generate(exp);
    pcode("", "=", expVar, "", id);
    return expVar;
   else
    op1 = node.childs[1].token;
    pcode("", op1[0], id.value, "1",
           id.value)
    return id;
   end if
else if (node.tag=COND)
   expVar1 = generate(node->childs[0])
   op = node->child[1];
   expVar2 = generate(node->childs[2]);
   pcode("", "CMP", expVar1, expVar2, "");
   return op.value
```

```
處理 STMT 陳述
id = EXP | id [++|--]
如果 id 之後為等號, id=EXP
   取得 EXP 節點
   產生 EXP 的程式
   產生指定敘述 (= T0 sum)
   傳回臨時變數 (T0)
否則, id [++|--]
   取得運算碼 (++ 或 --)
   輸出運算指令
      (+ i 1 i)
   傳回運算變數 (i)

處理布林判斷式 COND =
   EXP
   [==|!=|<=|>=|<|>]
   EXP
   輸出比較指令 (CMP i 10)
   傳回比較運算 (例如 <=)
```

圖 8.13 中間碼產生的演算法(第3頁)

```
  else if node.tag in [EXP]
    item1 = node.childs[0];
    var1 = generate(item1);
    for (ti=1; ti<node.childs.Count; ti+=2)
     op = node.childs[ti].token;
     item2 = node.childs[ti + 1];
     var2 = generate(item2);
     tempVar = nextTempVar();
     pcode("", op, var1, var2, tempVar);
     var1 = tempVar;
    end for
    return var1;
  else if (node.tag in [number|id])
    return node.token;
  else if (node.childs != null)
    foreach (child in node.childs)
     generate(child);
    return null;
  else return null;
  end if
End Algorithm
```

```
處理算式 EXP=ITEM([+-*/]ITEM)?
  取得 ITEM
  產生第一個運算元的程式
  針對後續的([+-*/]ITEM)?
    取得 [+-*/]
    取得 ITEM
    產生 ITEM 的中間碼
    取得新的臨時變數
    輸出運算指令(+ sum i T0)
    設定新臨時變數為傳回值

  傳回結果 (T0)
遇到變數或常數
  傳回其 token 名稱
針對其他狀況，若有子代
  則對每個子代
    遞迴產生程式
  不傳回值
否則，不傳回值
```

圖 8.13 中間碼產生的演算法(第4頁)

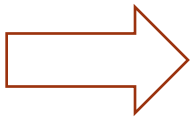
```
Algorithm pcode
Input label, op, params
  if (label is not empty)
    output label+":"
  output op, param[0], param[1], param[2]
End Algorithm
```

演算法 pcode()
輸入：標記、運算、參數
如果有標記
　就輸出標記到中間檔
輸出中間碼

8.6 組合語言產生

- 時機
  - 一旦中間碼產生完畢, 程式就可以輕易的將中間碼轉換成組合語言。
- 方法
  - 將中間碼指令轉為組合語言指令
  - 必須考慮佔存器的載入與儲存
- 範例

+ sum i T0



```
LD      R1      sum
LD      R2      i
ADD     R3      R1      R2

ST      R3      T0
```

範例8.4 將中間碼轉換為組合語言

▶範例 8.4 將中間碼轉換為組合語言的範例

(a) 中間碼

```
1       =   0       sum
2
3       =   0       i
4
5
6
7   FOR0:
8
9
10          CMP     i   10
11          J   >       _FOR0
12
13
14          +   sum i   T0
15
16
17          =   T0      sum
18
19
20          +   i   1   i
21
22          J           FOR0
23      _FOR0:
24
25          RET         sum
26
27
28
```

(b) 組合語言 (無最佳化)

```
    LDI   R1   0
    ST    R1   sum
    LDI   R1   0
    ST    R1   i


FOR0:
    LD    R1   i
   LDI   R2   10
   CMP   R1   R2
   JGT   _FOR0
   LD    R1   sum
   LD    R2   i
   ADD   R3   R1 R2
   ST    R3   T0
   LD    R1   T0
   ST    R1   sum
   LD    R1   i
   LDI   R2   1
   ADD   R3   R2 R1
   ST    R3   i
   JMP   FOR0
_FOR0:
   LD    R1   sum
   RET
sum:   RESW   1
i:     RESW   1
T0:    RESW   1
```

圖 8.14 將中間碼轉換為 CPU0 組合語言的演算法

▶圖 8.14 將中間碼轉換為 CPU0 組合語言的演算法

中間碼轉組合語言的演算法	說明
Algorithm pcodeToAsm	將 pcode 轉換為組合語言
Input label, op, p1, p2, p3	輸入：label 標記、op 運算、p1:參數 1…
if (label is not empty)	如果有標記 (例如 FOR0)
output(label)	輸出標記
if (op is "=")	如果是指定運算 = (例如 = T0 sum)
rewrite(LD R1, p1)	(例如：輸出 LD R1, T0)
rewrite(ST R1, p3)	(例如：輸出 ST R1, sum)
else if (op in [+-*/])	如果是加減乘除 (例如：+ sum i T0)
rewrite(LD R1, p1)	(例如：輸出 LD R1, sum)
rewrite(LD R2, p2)	(例如：輸出 LD R2, i)
rewrite(ASM(op), R3, R1, R2)	(例如：輸出 ADD R3, R1, R2)
rewrite(ST R3, p3)	(例如：輸出 LD R1, sum)
else if (op is CMP)	如果是 CMP 比較 (例如：CMP i 10)
rewrite(LD R1, p1)	(例如：輸出 LD R1, i)
rewrite(LD R2, p2)	(例如：輸出 LDI R2, 10)
rewrite(CMP R1, R2)	(例如：輸出 CMP R1, R2)
else if (op is J)	如果是跳躍 (例如：J > _FOR0)
jop = AsmJumpOp(op, p1);	(例如：將 J > 改為 JGT)
rewrite(jop, p3)	(例如：輸出 JGT _FOR0)
else if (op is RET)	如果是 RET (例如：RET sum)
rewrite(LD R1, p3)	(例如：輸出 LD R1, sum)
rewrite(RET)	(例如：輸出 RET)
End Algorithm	

指令的改寫

- 對常數值的載入指令改寫為 LDI

```
Algorithm rewrite(op, p1, p2, p3)
  if (op is LD) and isNumber(p2)
     op = LDI
  output(op, p1, p2, p3)
End Algorithm
```

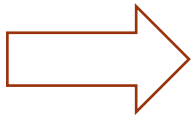
```
(例如：LD R2, 10 改為 LDI R2, 10)
如果 op 是 LD 且 p2 是整數
      將 op 改為 LDI
      輸出該指令
```

8.7 最佳化

中間碼

+ sum i T0

轉換

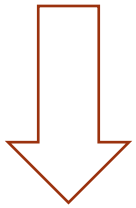


組合語言

```
LD      R1      sum
LD      R2      i
ADD     R3      R1      R2

ST      R3      T0
```

當 sum 已經在 R1 中，
i 已經在 R2 中，
而且 T0 對應到 R3 時



最佳化

```
ADD     R3      R1      R2
```

範例8.5 最佳化的範例

▶範例 8.5 最佳化的範例

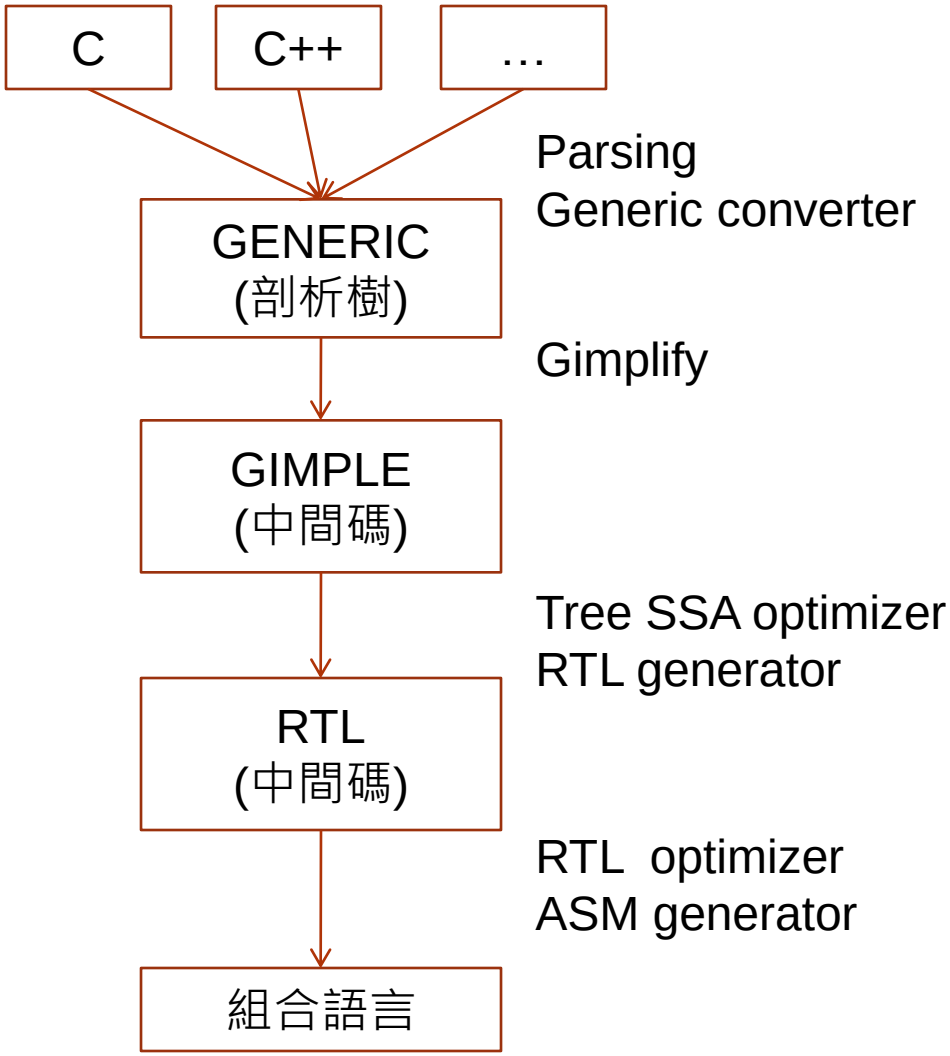
	(a) 中間碼	(b) 組合語言 (無最佳化)	(c) 組合語言 (有最佳化)
1	= 0 sum	LDI R1 0	LDI R1, 0
2		ST R1 sum	ST R1, sum
3	= 0 i	LDI R1 0	LDI R2, 0
4		ST R1 i	ST R2, i
5			LDI R3, 1
6			LDI R4, 10
7	FOR0:	FOR0:	
8		LD R1 i	
9		LDI R2 10	
10	CMP i 10	CMP R1 R2	CMP R2, R4
11	J > _FOR0	JGT _FOR0	JGT _FOR0
12		LD R1 sum	
13		LD R2 i	
14	+ sum i T0	ADD R3 R1 R2	ADD R1, R1, R2
15		ST R3 T0	
16		LD R1 T0	
17	= T0 sum	ST R1 sum	
18		LD R1 i	
19		LDI R2 1	
20	+ i 1 i	ADD R3 R2 R1	ADD R2, R2, R3
21		ST R3 i	
22	J FOR0	JMP FOR0	JMP _FOR0
23	_FOR0:	_FOR0:	
24		LD R1 sum	ST R1, sum
25	RET sum	RET	RET
26		sum:RESW 1	i RESW 1
27		i: RESW 1	sum RESW 1
28		T0: RESW 1	

8.8 實務案例：gcc 編譯器

- gcc 編譯器的過程

  - C 語言
    → (剖析) → generic 語法樹
    → (產生) → gimple 中間碼
    → (語意分析) → RTL 中間碼
    → (最佳化) → (組合語言產生)
    → 組合語言

圖 8.15 GNU 編譯器的流程



範例 8.6 gcc 編譯器的中間碼格式

▶範例 8.6 gcc 編譯器的中間碼格式

(a) C 語言

```
if (foo(a+b, c)) {
    c = b++ / a;
}
return c;
```

(b) Generic 中間碼

```
if (foo (a + b, c))
    c = b++ / a
endif
return c
```

(c) Gimple 中間碼

```
    t1 = a + b
    t2 = foo (t1, c)
    if (t2 != 0) <L1, L2>
L1:
    t3 = b
    b = b + 1
    c = t3 / a
    goto L3
L2:
L3:
    return c
```

範例 8.7 中間碼 Gimple 與 RTL 之對照範例

- 將 Gimple 轉為 RTL 的範例

▶範例 8.7 中間碼 Gimple 與 RTL 之對照範例

(a) Gimple	(b) RTL	(c) 簡化後的 RTL
b = a - 1	(set (reg/v:SI 59 [b]) (plus:SI (reg/v:SI 60 [a] (const_int -1 [0xffffffff]))))	b (59, reg/v:SI) = a (60, reg/v:SI) + -1 (const_int)

- 8.7(b) RTL 範例的解讀

```
b (59, reg/v:SI) = a (60, reg/v:SI) + -1 (const_int)
```

要求 gcc 編譯器輸出 rtl 中間碼

- 指令：
  - 加上 –dr 參數, 可讓 gcc 輸出 rtl 中間碼

指令與操作過程	說明
C:\ch08>gcc -c -dr sum.c -o sum.o	用-dr 參數編譯 要求輸出中間碼
C:\ch08>dir	
…	
2010/03/12 上午 09:00 105 sum.c	
2010/04/09 上午 09:29 3, 784 sum.c.01.rtl	RTL 中間碼檔案
2010/04/09 上午 09:29 372 sum.o	
3 個檔案 4, 261 位元組	
3 個目錄 9, 196, 593, 152 位元組可用	

RTL 檔案的內容

▶範例 8.8 C 語言與其 RTL 片段

(a) C 語言程式

```
int sum(int n) {
   int s=0;
   int i;
   for (i=1; i<=n;i++) {
       s = s + i;
   }
   return s;
}
```

(b) 對應的 RTL 檔案

```
(note 2 0 3 NOTE_INSN_DELETED)
...
(insn 8 6 11 (set (mem/f:SI (plus:SI
   (reg/f:SI 54 virtual-stack-vars)
   (const_int -4 [0xfffffffc])) [0 s+0 S4 A32])
   (const_int 0 [0x0])) -1 (nil)
   (nil))
...
(jump_insn 16 15 17 (set (pc)
   (if_then_else (gt (reg:CCGC 17 flags)
      (const_int 0 [0x0]))
      (label_ref 30)
      (pc))) -1 (nil)
   (nil))
...
```

Gcc 的最佳化功能

範例 8.9 gcc 不同層級的最佳化實例

編譯指令(無最佳化)： gcc -S optimize.c -o optimize_O0.s -O0
編譯指令(O3 級最佳化)： gcc -S optimize.c -o optimize_O3.s -O3

(a) optimize.c

```
int f() {
  int a=3, b=4, c, d;
  c=a+b;
  d=a+b;
  return c+d;
}
```

(b) optimize_O0.s (無最佳化)

```
  .file  "optimize.c"
  .text
.globl _f
  .def   _f;  .scl    2;
  .type  32;  .endef
_f:
  pushl  %ebp
  movl   %esp, %ebp
  subl   $16, %esp
  movl   $3, -4(%ebp)
  movl   $4, -8(%ebp)
  movl   -8(%ebp), %eax
  addl   -4(%ebp), %eax
  movl   %eax, -12(%ebp)
  movl   -8(%ebp), %eax
  addl   -4(%ebp), %eax
  movl   %eax, -16(%ebp)
  movl   -16(%ebp), %eax
  addl   -12(%ebp), %eax
  leave
  ret
```

(c) optimize_O3.s (有最佳化)

```
  .file  "optimize.c"
  .text
  .p2align 4, , 15
.globl _f
  .def   _f;  .scl    2;
  .type  32;  .endef
_f:
  pushl  %ebp
  movl   $14, %eax
  movl   %esp, %ebp
  popl   %ebp
  ret
```

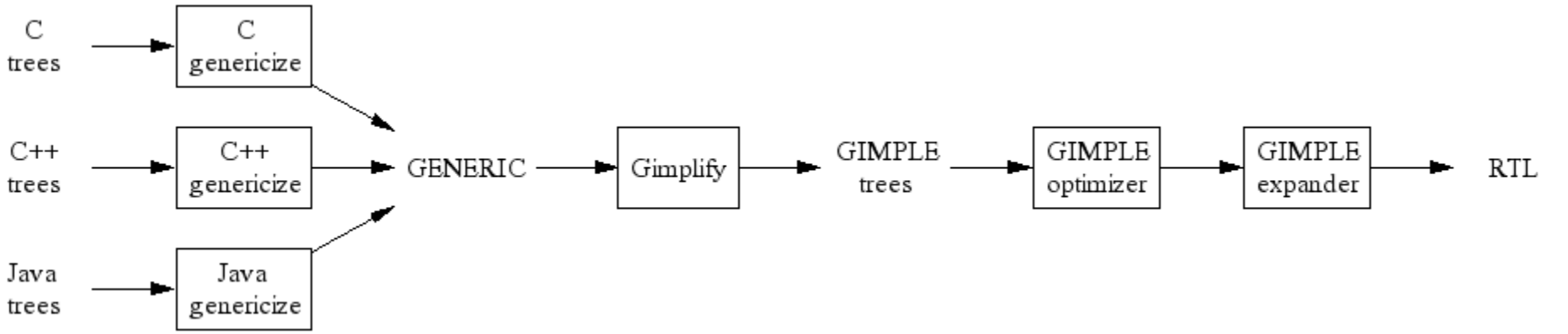
結語

- 高階語言
  → (掃描) → 詞彙串列
  → (剖析) → 語法樹
  → (語意分析) → 語意樹
  → (中間碼產生) →中間碼
  → (組合語言產生) → 組合語言
  → (組譯器)
  → 目的檔「機器碼」

習題

- 8.1 請為 C0 語言加上 if 語句的 EBNF 語法, 加入到圖 8.2 中。
- 8.2 接續上題, 請在圖 8.9 當中加入剖析 if 語句的演算法。
- 8.3 接續上題, 請在圖 8.13 當中加入將 if 語句轉為中間碼的演算法。
- 8.4 請為範例 8.5 (b) 的無最佳化組合語言, 提出一個簡單的最佳化機制, 並寫出您的最佳化方法實施後, 所產生的組合語言程式碼。
- 8.5 請使用 gcc 的 -O0 與 -O3 參數, 分別已無最佳化與高級最佳化的方式, 編譯任意一個 C 語言程式為組合語言, 並觀察其編譯後 的組合語言, 指出最佳化後哪些指令被省略了

其他圖片，不包含在書當中的


C
trees
C
genericize
C++
trees
C++
genericize
Java
trees
Java
genericize
GENERIC
Gimplify
GIMPLE
trees
GIMPLE
optimizer
GIMPLE
expander
RTL

Gimple

GENERIC

```
if (foo (a + b,c))
  c = b++ / a
endif
return c
```

High GIMPLE

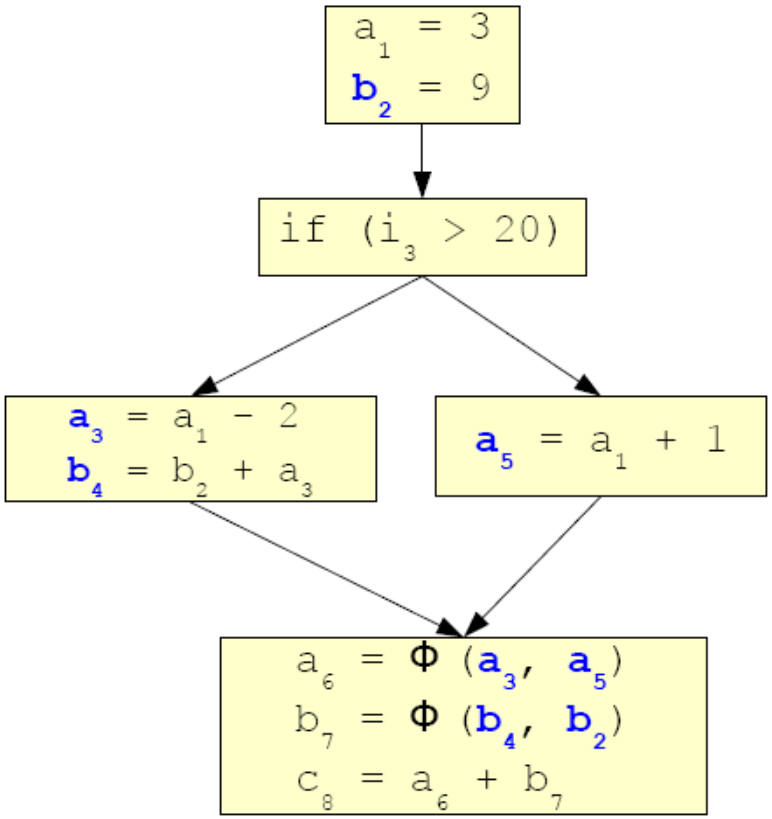
```
t1 = a + b
t2 = foo (t1, c)
if (t2 != 0)
  t3 = b
  b = b + 1
  c = t3 / a
endif
return c
```

Low GIMPLE

```
t1 = a + b
t2 = foo (t1, c)
if (t2 != 0) <L1,L2>
L1:
t3 = b
b = b + 1
c = t3 / a
goto L3
L2:
L3:
return c
```

Static Single Assignment (SSA)

- Versioning representation to expose data flow explicitly
- Assignments generate new versions of symbols
- Convergence of multiple versions generates new one (Φ functions)



較抽象的編譯器定義方式

圖 8.3 編譯器的三大階段 – 掃描、剖析、程式碼產生

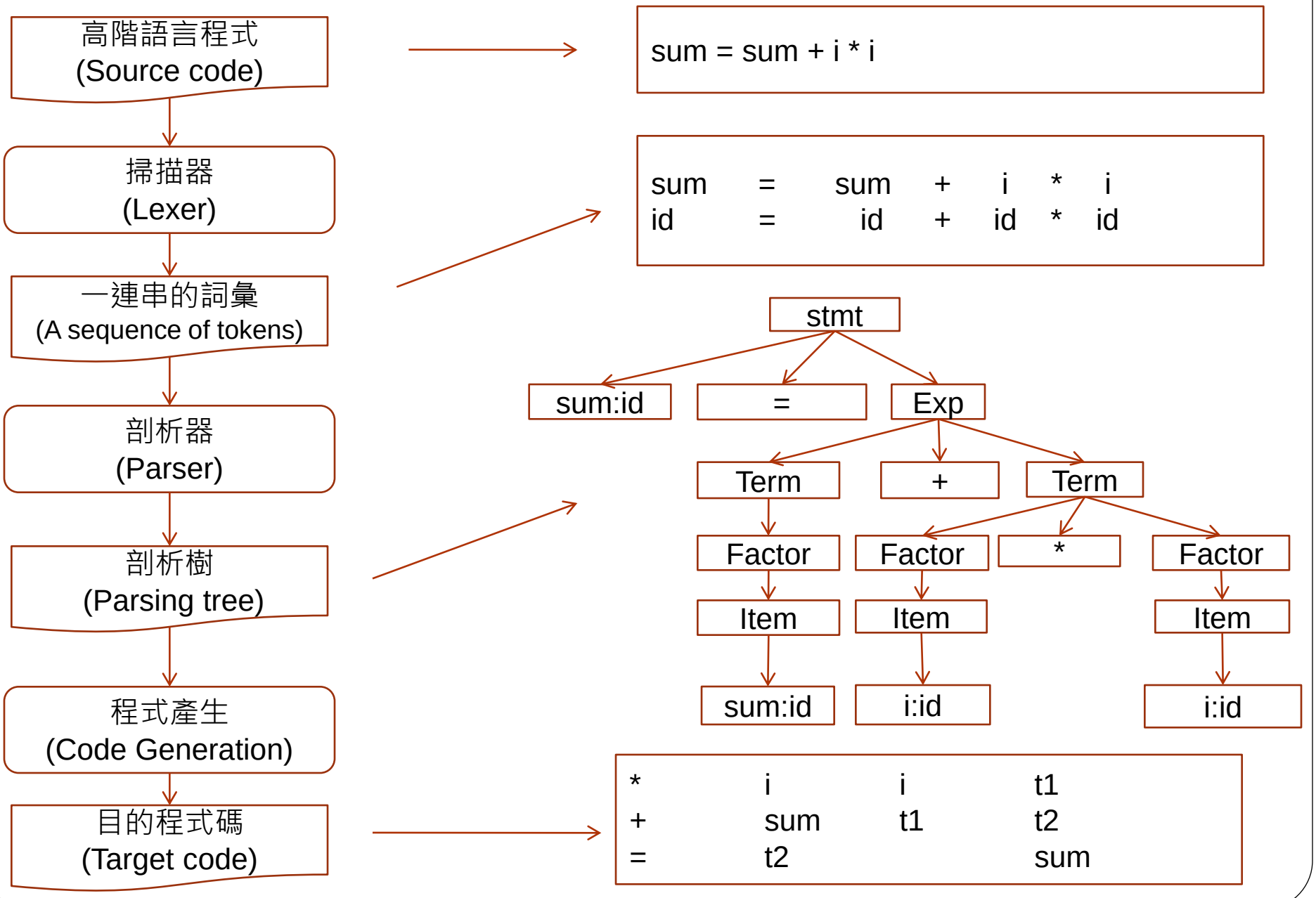


圖 8.4 編譯器的六階段模型

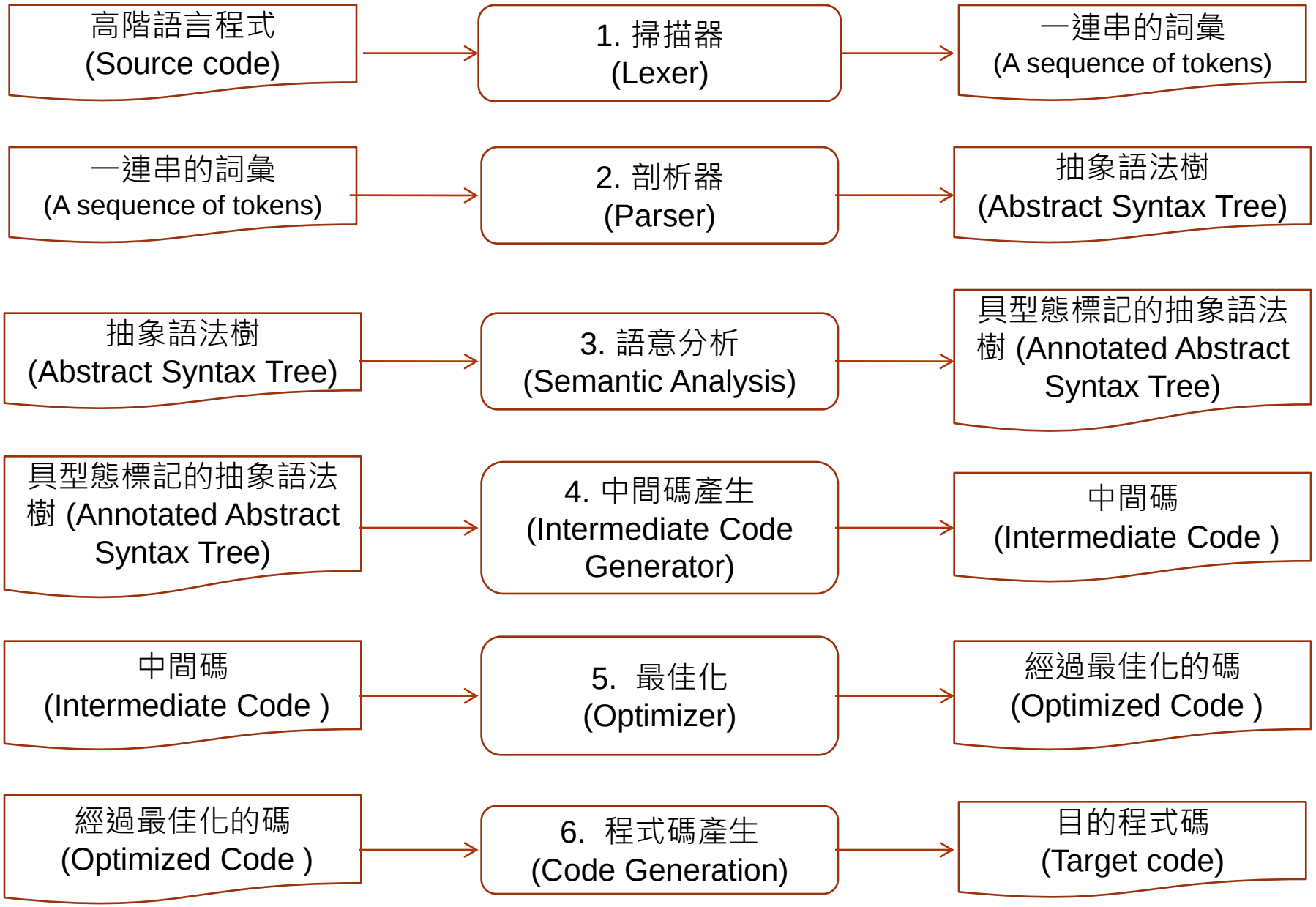
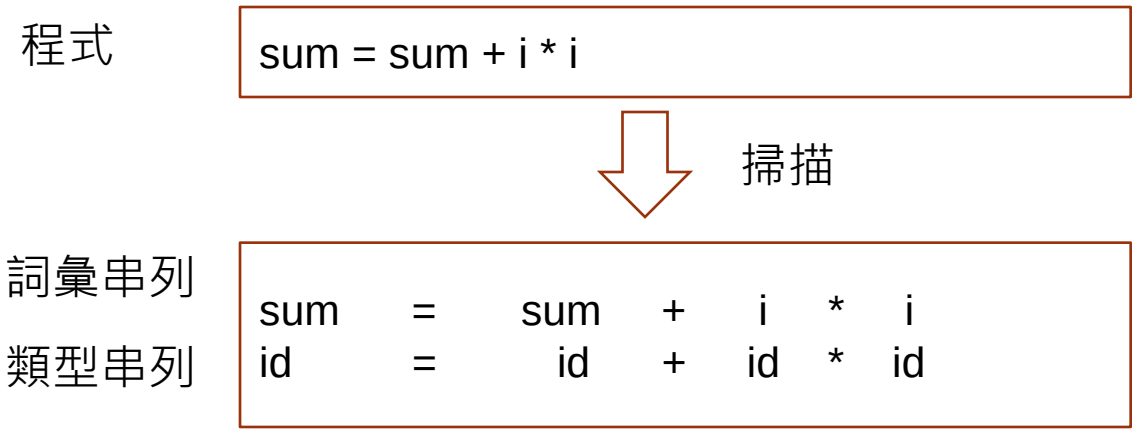


圖 8.5 階段一、掃描的過程示意圖



階段二、剖析的過程示意圖

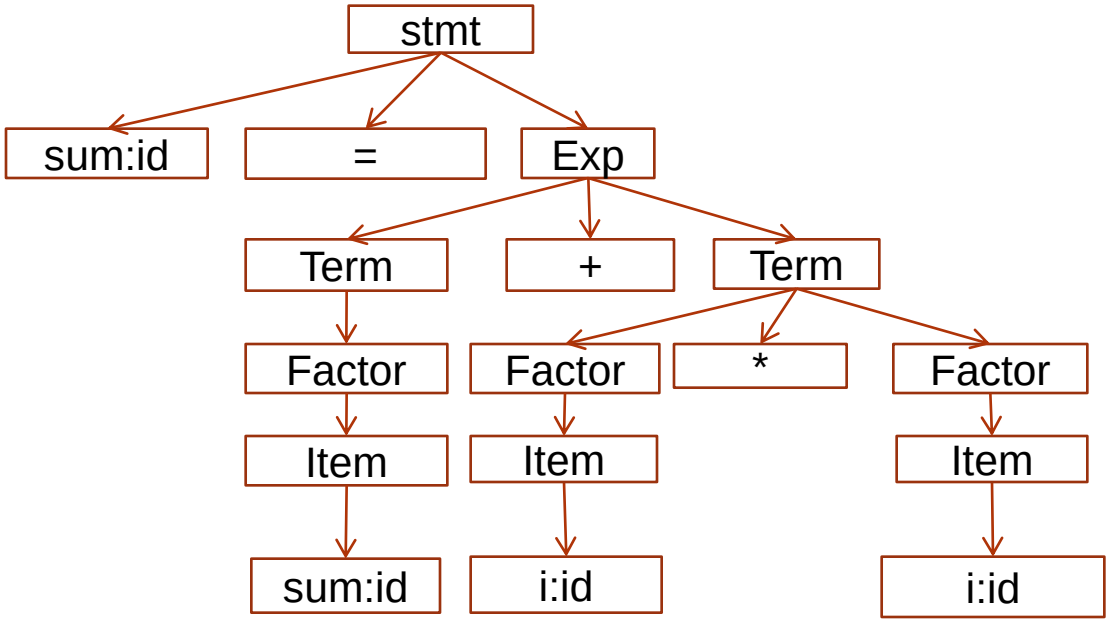
詞彙串列

類型串列

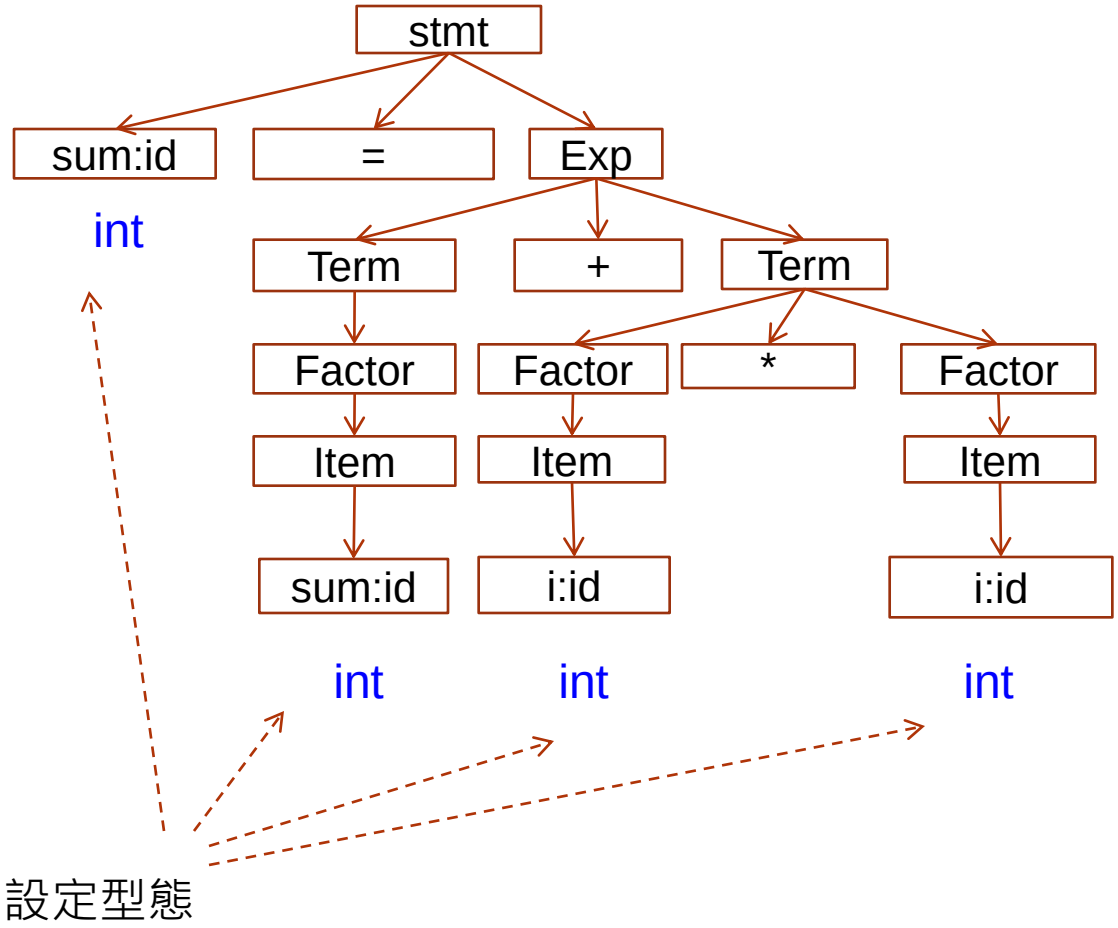
sum	=	sum	+	i	*	i
id	=	id	+	id	*	id

剖析

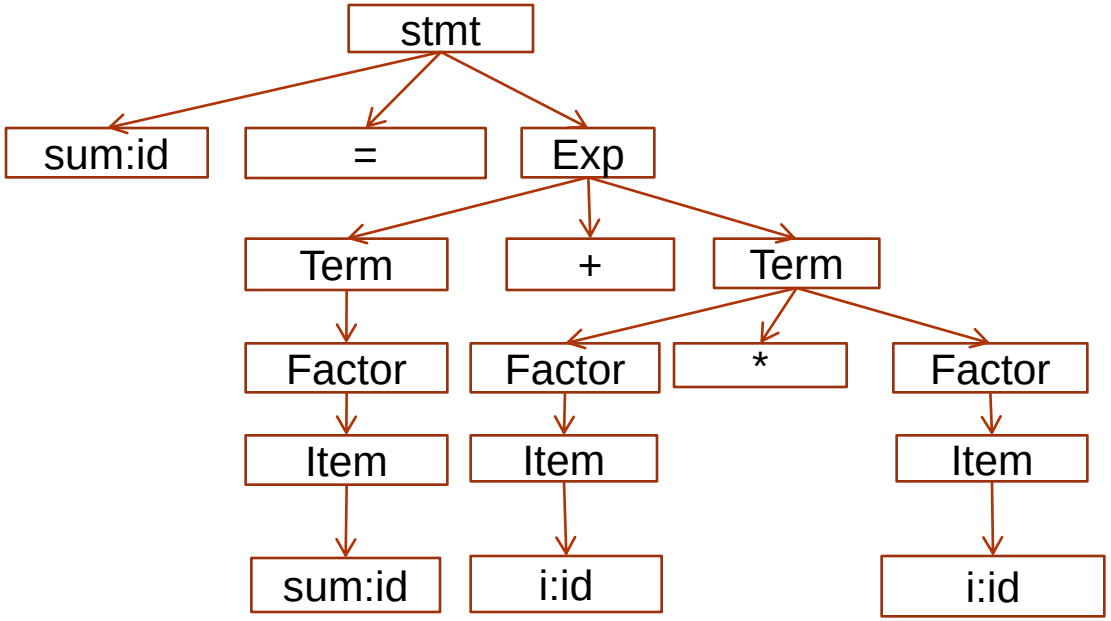
剖析樹



語意分析的過程示意圖


stmt
sum:id
=
Exp
int
Term
+
Term
Factor
Factor
*
Factor
Item
Item
Item
sum:id
i:id
i:id
int
int
int
設定型態

中間碼產生過程的示意圖



中間碼產生

*	i	i	t1
+	sum	t1	t2
=	t2		sum

階段五、最佳化過程的示意圖

*	i	i	t1
+	sum	t1	t2
=	t2		sum

最佳化

*	i	i	t1
+	sum	t1	sum

階段六、程式碼產生過程的示意圖

Pcode
中間碼

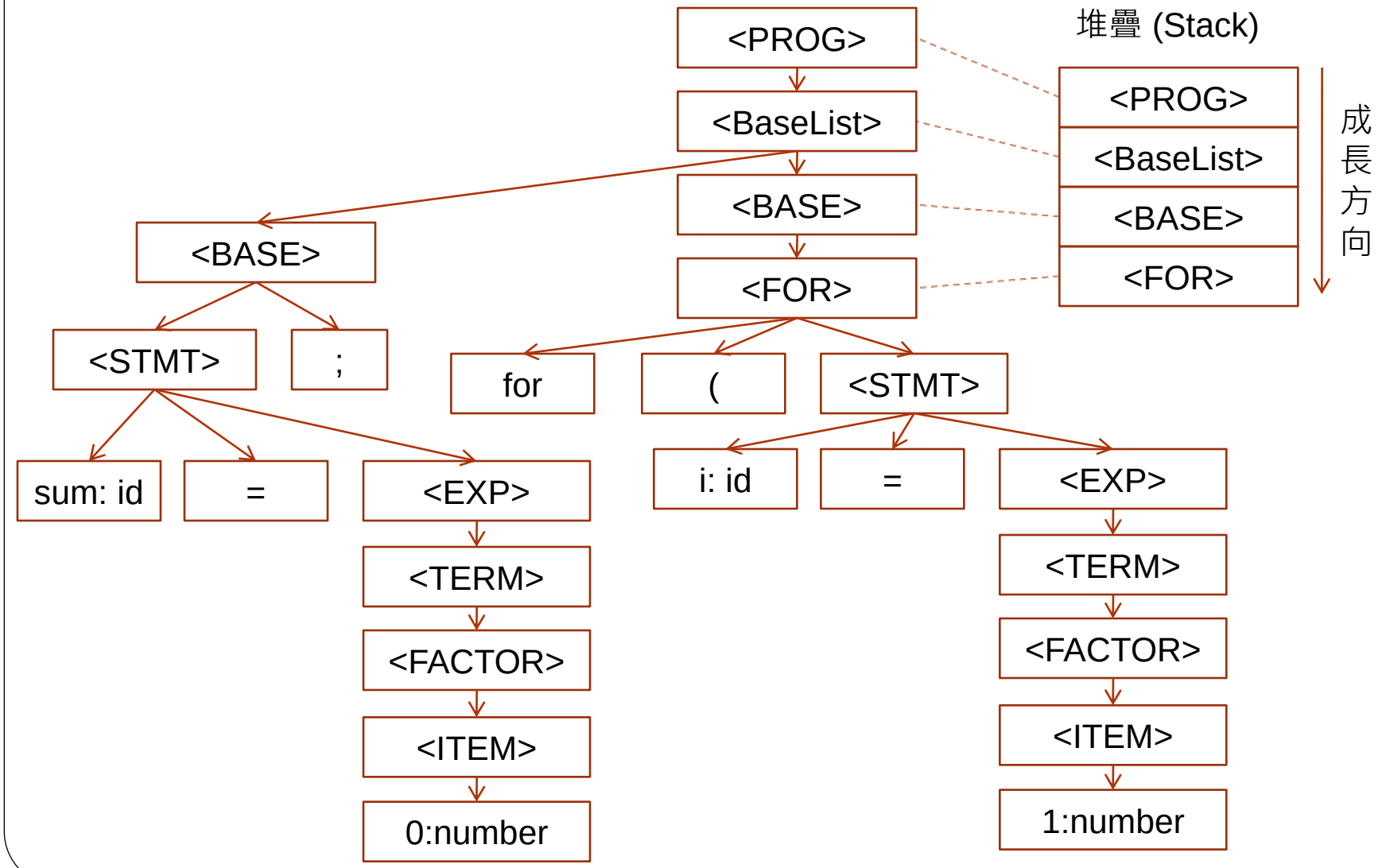
```
*       i       i       t1
+       sum     t1      sum
```

程式碼產生

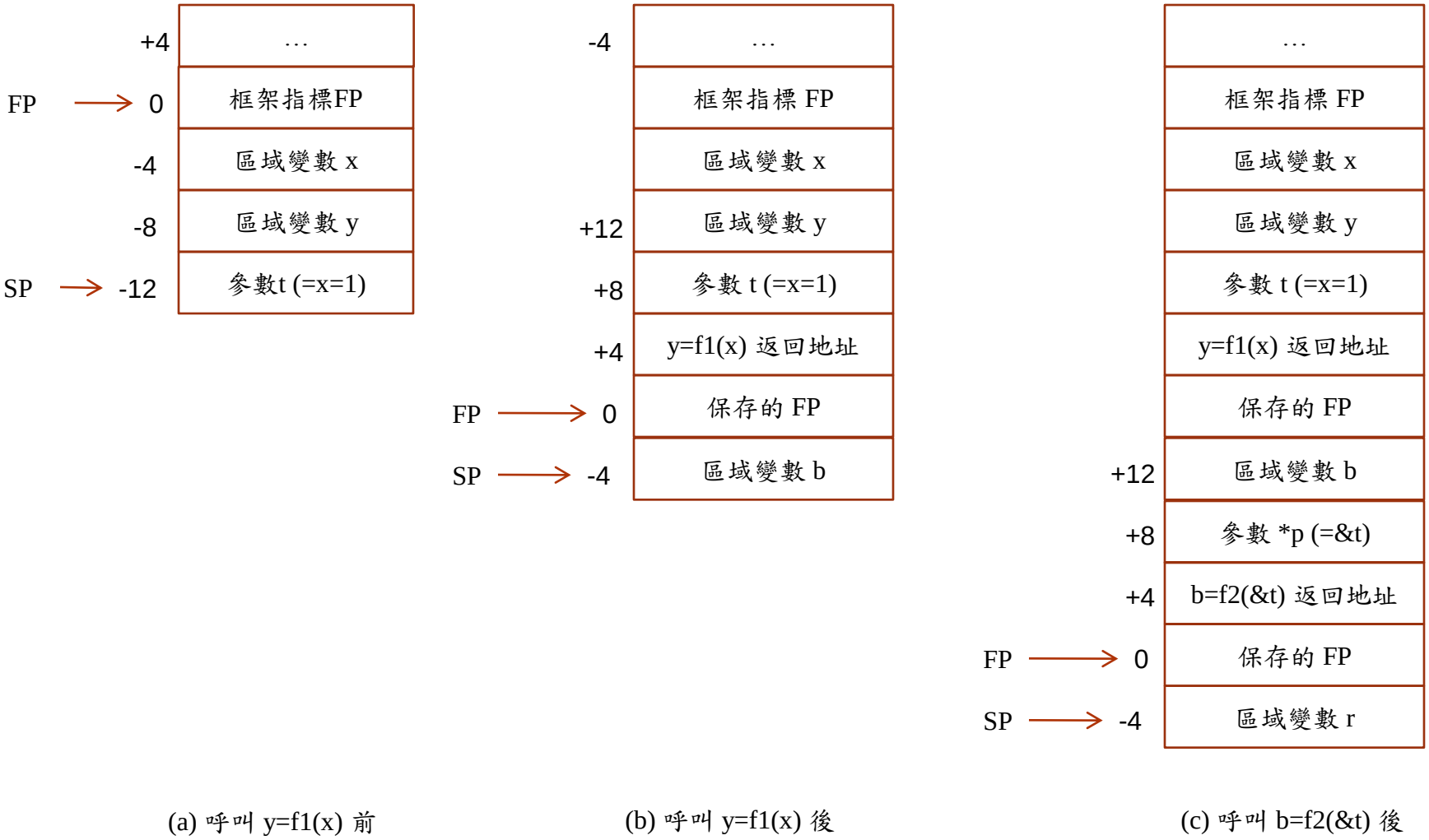
CPU0
組合語言

```
LD      R1, i
LD      R2, sum
MUL     R3, R1, R1
ADD     R2, R3, R2
ST      R2, sum
```

程式剖析到 i=1 完成時的語法樹與堆疊結構



副程式呼叫時的堆疊變化情況



(a) 呼叫 y=f1(x) 前　　(b) 呼叫 y=f1(x) 後　　(c) 呼叫 b=f2(&t) 後

註：在 CPU0 當中，返回值都是儲存在 R1返回

圖 8.20 <範例 8.4 > 程式的堆疊變化情況

使用gcc跨平台編譯的範例圖