1

開始學習 SKILL

1. 簡介

多年以來，CADENCE 公司的 CAD tool 一直是世界上使用最廣泛，功能最強大的積體電路(Integrated Circuit)設計工具。而爲了因應 IC 的複雜度越來越高，設計的困難度也越來越高的情況，CADENCE 的 CAD 整合開發環境也越來越龐大，所提供的功能也日益強大，造成使用者在維護及管理上的一大負擔。再則，每一家的設計公司的在設計的流程中多多少少都會有一些小步驟，無法用 CADENCE tool 提供的基本做法來達成； 或者是不同公司的 tools 之間資料轉換的問題。 工程師遇到此類問題可能需要透過人工的方式去完成連接設計流程中相連的兩個步驟； 或是乾脆去開發一些小軟體來完成這些特定的工作，而此時使用者可能會面臨如何將自己開發的軟體的 I/O 與 CADENCE tool 的整合環境相連結的問題。 一般的做法是產生一些資料檔來做資料交換的中介，這些資料檔的格式可能是 CADENCE 支援的一標準資料交換格式，也可以是使用者自訂的資料格式。這樣子是一種間接的做法，因爲使用者無法直接去存取 CADENCE 環境的內部資料，所以在處理上的彈性會小很多，也較不方便。

爲了方便使用者使用整個 CADENCE tools 的整合開發環境，以解決上述的困擾，CADENCE 公司遂發展了 SKILL 語言。 SKILL 是一種高階的、交談式的語言，是用於 CADENCE tool 的整合開發環境內的命令語言（command language）。 SKILL 採用人工智慧語言 LISP 的語法爲藍本，再加上常用的 C 語言的部份語法設計而成。SKILL 語言提供許多的介面函式，能讓使用者可以撰

寫程式直接去存取 CADENCE 整合環境內的電路資料內容； 也可以讓使用者去開發將自己開發的應用程式併入 CADENCE tool 的整合環境裡。有了 SKILL 言，使用者可以讓 CADENCE tool 更充份地融入整個設計流程之中，減少瑣碎的人工轉換時間，提升公司的生產力。

1.1 LISP 式的語法

在 SKILL 裡面，使用函式的呼叫方式可以有兩種方式：

（一）Algebraic 表示形式，也就是

Func（arg1 arg2 …）

（二）前置表示形式，此爲 LISP 型式的語法

（Func arg1 arg2 …）

程式是由敘述來組成的，正如在 LISP 語言裡面一樣，SKILL 的敘述是以串列（list） 的形式來表示。 如此的設計方式使得程式可以和資料用同樣的方式來處理。 使用者可以動態地建立、修改、或計算函式或表示式的值。

另外 ，在 SKILL 中不像一般的程式語言一樣有提供字元這種資料形態，字元就是用符號本身來表示 ，例如字元 “A”就是用”A”這個符號（變數）來代表。

2. 快速瀏覽 SKILL

2.1 解釋名詞

本節將介紹一些專有名詞：

名詞	意義
OUTPUT	SKILL 程式執行結果可以顯示到 xterm、設計視窗、檔案、或是命令解譯視窗 CIW（command interpreter window）
CIW	很多 CADENCE 的應用程式之起始工作視窗，包含有一行命令輸入列、一個輸出區域、一個功能選單列。
SKILL expression	呼叫一個 SKILL function 的程式敘述
SKILL function	一個有命名的、參數化的程式段

要啓動一個 SKILL 的 function 有幾種方式，在不同的 CADENCE 的應用程式裡，使用者可以透過 Bindkey, Form, Menu, 或 SKILL process 等來啓動，其意義如下：

名詞	意義
Bindkeys	將一個 function 與一個鍵盤的事件關聯起來，當使用者引發一個鍵盤事件時，CADENCE 軟體便會計算其相關聯的 function
Form	有些函式需要使用者提供一些資料輸入，通常是透過一個跳出式表格（pop-up form）來輸入資料
Menu	當使用者選擇 menu 上的一個項目時 ，系統便會執行相關之 SKILL 函式的計算
CIW	使用者可以直接在 CIW 輸入一個 SKILL function 來得到一個立即的計算結果
SKILL process	使用者也可以透過一個 UNIX 環境底下的行程，來啓動 CADENCE 裡的 interpreter，以便直譯某些 SKILL 程式

所有的 SKILL 函式都會傳回一值。在本書中我們將用“⇨”來表示函式的回傳值。 在 SKILL 裡面，大小寫是不同的。 要呼叫一個 SKILL 的函式的方式如下：

```
strcat( ”How” ”are” ”you” )
⇨”How are you”
```

或者用：

```
( strcat “How” “are” “you” )
⇨”How are you”
```

注意的是，在函式名稱與左括弧之間不可以留空白。 函式的內容可以分成幾行來寫，不一定要在同一行才可以。同樣地，幾個函式也可以放在同一行上，但此時只有最後一個函式值會回傳到螢幕上。

2.2 基本資料型態

在 SKILL 裡面資料也是大小寫不一的（case-sensitive），最簡單的資料型態有整數、浮點數、與字串。字串用雙引號 “” 來括起來。

2.3 運算子

SKILL 提供了一組運算子，每一個運算子對應到一個 SKILL 的函式。下表是一些常用的運算子：

運算子（依優先順序由大到小排列）	對應之 SKILL 函式	運算
**	expt	Arithmetic
*	times	“
/	quotient	“
+	plus	“
-	difference	“
++S, S++	preincrement, postincrement	“
= =	equal	tests for equality
! =	nequal	tests for equality
=	setq	Assignment

以下是一個應用運算子的例子：

x =5 y =6x+y

⇨11

2.4 變數

在 SKILL 的程式中你不需要去宣告變數， 一個變數的名稱可以包含英文字母、數字、底線（__）、問號（？）。 變數名稱的第一個字不可是數字。你可以用= 運算去指定一個變數的值。 你也可以直接鍵入一個變數名稱來執行以取得其值。你也可以用 type 這個函式來取得變數的資料型態。

3. SKILL 串列

SKILL 串列 簡單地說就是一些依序排列的 SKILL 資料物件。一個串列內可以包含任何 SKILL 資料型態的資料，一個串列的資料表示式要用括弧括起來。SKILL 允許有空的串列，表示成“（）”或 *nil*。 串列 裡面的元素也可以是另一個串列。以下是一些串列的例子：

串列	說明
（1 2 3）	一個包含１, 2, 3 三個整數的串列
（1）	一個包含 一個整數 1 的串列
（）	一個空的串列
（1（2 3）4）	一個串列 裡面還包含另一個串列

至如何指定一個串列變數的值呢？舉例如下：

S1= ‘(”A” ”B” “C”)

請注意在左括號之前須加上一個單引號。

3.1 建立串列

有幾個方法可以建立一個串列，舉例如下：

1. 直接使用單引號，例如：
   ‘(２４６) ⇨(２４６)
2. 呼叫使用串列 函式，例如：
   x=2 y=4
   list(x y 6) ⇨(２４６)
3. 使用 cons 函式，例如：
   r = ‘(4 6)
   r =cons(2 r) ⇨ (２４６)
4. 用 append 函式來合併兩個串列：
   alist = ‘(１2 3)
   blist = ‘(4 5 6)
   clist =append（alist blist） ⇨(１２3４5６)

3.2 串列與座標表示

由串列衍生出來的實際資料表示有兩個最主要的是座標（coordinates）與界限方塊（bounding box）。 對佈局圖（layout）而言， 吾人常用到的是這兩樣東西。在 SKILL 中 xy 座標可用兩個元素的串列來表示，例如

xVal=100

yVal=200
pCoordinate= xVal:yVal ⇨（100 200）

而 bounding box 是用左下及右上兩點座標來表示，例如

bBox=list (300:400:500:600)

或者可以用下列方式來產生：

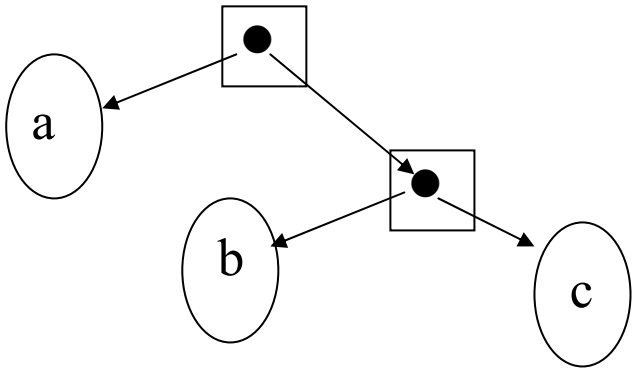
lowL = 300:400
upperR=500:600
bBox=list(lowerL upperR)

或者直接用雙層串列

bBox= ‘((300 400) (500 600))

3.3 使用串列

在 SKILL 中， 一個串列的內部儲存方式相當於是二元樹（binary tree）的結構，每一個內點都相當於一個分支決定點（branch decision point），例如：有一個 list(a b c)，此一 list 在內部儲存的方式可以下圖表示



在 SKILL 語言中，提供了很多的 list 相關的函式，以下是較基本的：

函式名稱	作用	範例	
1Chapter 3 Car	取出 list 中第一個元素。也可取出 bBox 的左下點座標	A='(1 2 3 4) car（A）	⇨（1 2 3 4） ⇨ 1
Cdr	去掉 list 第一個元素之後的子 list	A='(1 2 3 4) car（A）	⇨（1 2 3 4） ⇨（2 3 4）
Nth	取出 list 中的第 n 個元素	A='(1 2 3 4) nth（3A）	⇨（1 2 3 4） ⇨ 3
member	判斷某個資料項目是否在串列中	A='(1 2 3 4) member（3A）	⇨（1 2 3 4） ⇨（3 4）
length	計算一個串列的元素個數	A='(1 2 3 4) member（3A）	⇨（1 2 3 4） ⇨（3 4）
xcoord	取出 x 座標值	pCoor=100:200 xCoord（pCoor）	⇨（100 200） ⇨ 100
ycoord	取出 y 座標值	pCoor=100:200 yCoord（pCoor）	⇨（100 200） ⇨ 200
cadr	右上點座標	pBox='((100 150)（200 500）) upperR=cadr（pBox）	⇨（200 500）
caar	左下點之 x 座標	lowLx=caar（pBox）	⇨ 100
cadar	左下點之 y 座標	lowLy=caar（pBox）	⇨ 150
caadr	右上點之 x 座標	upperRx=caadr（pBox）	⇨ 200
cadadr	右上點之 y 座標	upperRx=caadr（pBox）	⇨ 250

4.檔案輸出／輸入

本節介紹如何在 SKILL 程式裡面去存取一個 UNIX 底下的檔案， 並且介紹如何去產生一個特定格式的輸出檔案。

4.1 顯示資料

在 SKILL 的程式裡面可以使用 *print* 和 *println* 這兩個函式來顯示資料到螢幕上。其中 *println* 的函式會在輸出資料之後再加上一個“換行”的控制字元。例如：

print (“hello”) print(“hello”)

的結果是

“hello” ”hello”

而執行

println(”hello”) println(”hello”)

的結果是

“hello”

“hello”

另外可以用 *printf* 這個函式來產生格式化的輸出結果，其用法類似在 C 語言裡面的 *printf* 函式。我們用以下的例子來說明：

printf（”%-15s %-10d” layerName rectCount）

其中括號內的第一個引數是一個字串，稱爲控制字串。 在控制字串裡面的每一個%開頭到空白爲止的子字串，稱爲一個“指引”（directive），用以指示在控制字之後對應的資料將如何被顯示出來。

控制字串內的“指引”的文法型式如下

%【-】【寬度】【精確度】轉換碼

其中由中括號包含者表示是可有可無的。現在說明其意義：

代號	意義	
【-】	代表向左對齊	
【寬度】	資料顯示出來至少佔據的格數	
【精確度】	要顯示出來的資料字數	
轉換碼	整數:	d
	浮點數:	f
	字串／符號:	s
	字元:	c
	數值:	n
	串列:	L
	point 串列:	P
	bounding box 串列:	B

以下是另外一些例子：

```
aList = ’(5 10 15)
printf( ”\n A list: ”%L” aList)        ⇨t
A list: (5 10 15)
```

4.2 資料寫入檔案

要寫入資料到一個檔案有幾個步驟。首先，用 *outfile* 函式去取得一個檔案的輸出埠（output port）；其次，使用 *print*，*println*，或 *fprintf* 指令來寫入資料；最後再用 close 函式將輸出埠關閉。

以下是應的範例：

```
port1 = outfile( ”/temp/file1”)
println( list( ”a” ”b”) port1)
println( list( ”c” ”d”) port1)
close( port1)
```

結果在檔案 /temp/file1 內看到

（”a” ”b”）
（”c” ”d”）

不像 print 或 println 函式，fprintf 函式的輸出埠參數是放在第一個引數的位置。 其使用類似 printf 函式，舉例如下：

```
port1=outfile(”/temp/file1”)
I=10
fprintf(port1 ”A number:%d”  I)
close(port1)
```

結果在/temp/file1 裡面可以看到

```
A number:10
```

4.3 讀檔

要從檔案去讀取資料也有幾個步驟。首先，用 *inflie* 函式去取得一個檔案的輸入埠（input port）；其次，使用 *gets* 函式去一次讀取一行資料，或使用 fscanf 來讀取格式化的資料；最後再用 close 函式將輸入埠關閉。

以下是應的範例：

```
inport=infile (”/temp/file1”)
gets (nextLine inport)
println (nextLine)
close (inport）
```

注意，inport 是放在 gets 函數的第二個引數位置。接下來是 fscanf 的應用：

```
inport=infile(”/temp/file”)
fscanf ( inport”%s”  w)
println(w)
close(inport)
```

fscanf 的控制字串原理與前面介紹 printf 的類似。常用的轉換碼有 %d、 %f、與 %s，分別代表整數、浮點數、與字串。

5. 控制流程

就像所有的程式語言一樣， SKILL 也提供了一些流程控制的指令，以及關係運算子。在以下的小節中將會逐一介紹。

5.1 關係與邏輯運算子

在 SKILL 中的關係運算子如下表所列：

運算子	引數型態	對應之函式	範例	回傳值
<	numeric	lessp	1<2	t
			4<2	nil
<=	numeric	leqp	4<=6	t
>	numeric	greaterp	10 > 5	t
>=	numeric	geqp	3>=1	t
= =	numeric	equal	2.0= =2	t
	string		“xyz”= =”XYZ”	nil
	list			
!=	numeric	nequal	“xyz”!=”XYZ”	t
	string			
	list			

邏輯運算子主要有兩個：

運算子	引數型態	對應之函式	範例	回傳值
&&	General	and	1 && 2	2
			4 && 6	6
			a && nil	nil
			nil && t	nil
‖	General	or	3 ‖ 2	3
			5 ‖ 4	5
			a ‖ nil	t
			nil‖ a	t

5.2 if 函式

使用 if 指令大概是讀者最熟悉的控制命令了，幾乎所有程式語言都有。在 SKILL 裡面 if 是一個函式，其使用方法可由下面的例子得知

```
if（Shape= =”rect”
        then   println（”Rectangle”）
               count= count +1
        else
               println（”Not rectangle”）
               miscount= miscount + 1
）
```

注意！在 if 與左括弧之間不可以有空白，否則會出現錯誤訊息。

5.3 when 與 unless 函式

when 的用法如下例：

```
when(Shape= =”rect”
          println(”Rectangle”)
          ++count
        )；when
```

當 when 迴圈內的判斷式爲真時，則繼續執得迴圈內的敘述。而 unless 的用法也是當迴圈內的條件判斷式爲真時執行迴圈內的敘述。如下例：

```
unless(Shape= =”rect”)
          println(”Rectangle”)
          ++miscount
        )；unless
```

注意的是 *when* 與 *unless* 函式都會回傳值，亦即最後一回迴圈所計算出來的值或者是 nil。

5.4 case 函式

case 函式提供了多重分支（branching）的控制敘述，其用法如下例：

```
case( Letter
        （”a”   ++aCount   println（”A”）   ）
```

```
        （”b”   ++bCount   println（”B”）    ）
        （”c”   ++cCount   println（”C”）    ）
        （t     ++oCount   println（”Others”）
    ）；case
```

當上述 *case* 函式執行時，變數 *Shape* 的值會依序跟每一個分支部份（arm）的標記（label）來做比對（亦即”a”、”b”及”c”），如果相符則執行此一分支部份（arm）的敘述式。 如果所有的值都沒有匹配成功，則最後一標記爲 *t* 的分支部份的敘述會被執行。

如果有一分支部份的標記值是串列的話，SKILL 會逐一去比對串列裡面的元素，只要有一個比對成功，則此一分支部的敘述便會被執行。如下例：

```
case( Letter
        （”a”   ++aCount   println(”A”)                  ）
        （（”b” “c”）  ++bcCount   println(”B or C”)     ）
        （t   ++oCount   println(”Others”)               ）
    ）；case
```

只要 Letter 的值是”b”或”c”，第二個分支部的敘述都會被執行。

5.5 for 函式

for 函式提供迴圈控制之用，其使用的方式如下例：

```
R = 0
for( i   0   10
     R =R + i
     println(”The result is:” sum)
  )
⇨ t
```

其中的索引變數 *i* 的值在進入 *for* 迴圈時會被存起來，待離開迴圈之後再回復其值。在迴圈執行時 *i* 的值由 0 遞增到 10 ，每次增加 1。要注意 for 與左括弧 “（” 之間不可以留空白。

5.6 foreach 函式

foreach 函式提供一個很方便的方法來針對一個串列裡面所有的元素進行同樣的處理動作。下面是一個 foreach 的應用範例：

```
aCount = bCount = cCount = nCount = 0
letterList = ’(”a” “a” “c” “b” “c”)
```

```
foreach(   letter   letterList
      Case( letter
            (”a”     ++aCount )
            (”b”     ++bCount )
            (”c”     ++cCount )
          ) ;case
       ) ; foreach
```

執行時 foreach 函式時 letterList 裡面的元素會依序指定給 letter 變數，在每一個迴圈開始時 SKILL 會指定下一個 letterList 裡面的元素指定給 letter 變數

6.發展一個 SKILL 函式

要發展一個 SKILL 的函式需要幾項工作，分別在下列幾節中加以介紹。

6.1 群組化 SKILL 敘述

所謂將一堆 SKILL 敘述群組化的意思就是用左括號｛ 與右括號 ｝將一堆敘述包起來。 如此一來 SKILL 會將這堆敘述視爲一個大的敘述，而此一敘述執行時的回傳值就是裡面最後一個敘述執行的回傳值。 底下是一個集合敘述的例子：

```
bBoxHieght = {
     bBox = list( 200:250 250:400)
     lowLeft = car( bBox)
     upRight = cadr( bBox)
     lowLeftY = yCoord( lowLeft)
     upRightY = yCoord( upright)
     upRightY – lowLeftY   }
```

在這個例中，最後的回傳值就是最後一個敘述的值，我們將它指定給變數 *bBoxHeight*。

6.2 宣告一個 SKILL 函式

要以一個特定的名稱來參考一個敘述的群組，必須先用 procedure 的宣告來給定此一群組一個名稱。此一名稱加上敘述群組便構成一個 SKILL 的函式。此

一名稱爲函式名稱，而集合的敘述稱爲函式的本體（body），要執行一個函式必須使用函式名稱再加上（ ）。 同時，爲了使宣告的函式更有彈性，可以在函式名稱之後的小括弧內加入傳遞參數的宣告。 以下是一個函式宣告及呼叫函式的例子：

```
procedure(CalBBoxWidth( bBox)
        ll      =car( bBox)
        ur       =cadr( bBox)
        llx     =xCoord(ll)
        urx     =xCoord(ur)
        urx – llx
  ) ; procedure

bBox = list(100:100 200:300)
bBoxWidth= CalBBoxWidth(bBox)
```

2

SKILL 語言特徵

1. SKILL 文法

本章節介紹 SKILL 使用之特殊字元符號、註解之應用。

1.1 特殊字元

下表列出一些特殊字元及其意義：

字元	意義
\	供特殊符號跳脫原來用意之用
（）	群組串列
【】	陣列之索引，超級右括號（super right bracket）
{ }	群組一堆敘述
‘	在引用一個敘述時，防止其被執行計算
“	字串分隔字元
,	串列元素間的分隔字元（可有可無）
;	註解一行時的分隔字元
:	位元欄位界限字元
·	getq 之運算子
+-*/	進行算數運算

! ^ & |	進行邏輯運算
<>=	關係運算子
#	如果置於行首時，表示指引解譯器進行特定的解譯動作
@	如果是第一個字元，代表保留字的意思
?	如果是第一個字元，代表是關鍵字參數
`	引用一個表示式，而不去計算其值

1.2 註解

SKILL 提供了兩種的註解寫法。一種是區塊導向的寫法，如下例：

```
/* This is a block of comments */
```

另一種是整行的註解方式：

```
x =1                ; comment following a statemnet
; comment line 1
; comment line 2
```

在分號之後一直到行尾的部份都是註解。

1.3 其他

在使用 SKILL 語言時，有一些關於文法規則的細節是較特別的，在本節中特別將其歸納如下：

（1） 呼叫函式時，函式名稱與左括弧之間不得有空白。 所以 *f(x)*是合法的呼叫；*f (x)*是不合法的呼叫。

（2） 使用單元運算子如正、負號時，正負號與變數之間也不得有空白。所以 *–a* 是合法的，而 *– a* 是不合法的。相反的，使加、減號的二元運算子時，運算子的兩側必須同時有空白，或同時沒有。所以 *a-b* 或 *a – b* 是合法的；但 *a –b* 是不合法的。

（3） 不要用括號來包含一個變數或常數，例如 (1)， (*x*)，或用兩個小括弧去括一個表示式（(*a*+*b*)）。因爲 SKILL 會將其內容視爲函式的名稱。

（4） 在 SKILL 的敘述中可以用倒斜線來接一行的內容，舉例如下

```
str = ” This is a \
continuous line.”
⇨ “This is a continuous line.”
```

（5） 在 SKILL 程式中可以用倒引號來建立串列，而在建立串列時可以用 “,”

或是“，@”的建構子句來參數化串列中的元素。舉例如下：

b= 1
y= ‘(e f)
‘(a b c)　　⇨（a b c）
‘(a ,b c)　　⇨（a l c）
‘(a ,y c)　　⇨（a（e f）c）
‘(a ,@y c)　　⇨（a e f c）

2.1 資料特性

SKILL 支援的資料型態如下表所列：

資料型態	內部名稱	單一助憶字元
arry	arry	a
CADENCE database object	dbobject	d
floating-point number	flonum	f
any data type	general	g
linked list	list	l
integer or floating point number		n
user-defined type		o
I/O port	port	p
defstruct		r
symbol	symbol	s
symbol or character string		S
character string（text）	string	t
function object		u
window type		w
integer number	fixnum	x
binary function	binary	y

SKILL 中提供整數的二進位／八進位／十進位／十六進位的表示：

基底	前置符號	範例〔十進位值〕
Binary	0b or 0B	0b0011[3] 0b0010[2]
Octal	O	077[63] 011[9]
hexadecimal	0x or 0X	0x3f[63] 0xff[255]

SKILL 提供一組比例因子（scaling factor）來表示數值，在比例因子與數值之間不可以有空白。

字元符號	名稱	倍數	範例
T	Tera	10^{12}	10T[1.0e13]
G	Giga	10^{9}	10G[10e9]
M	Mega	10^{6}	10M[10e6]
K	Kilo	10^{3}	10K[10000]
%	Percent	10^{-2}	5%[0.05]
m	Milli	10^{-3}	5m[5.0e-3]
u	micro	10^{-6}	1.5u[1.5e-6]

SKILL 表示控制字元是用倒斜線加上該字元的 ASCII 碼而成，而有些常用的控制字元則有特定的字元來代表。詳列如下：

字元	跳脫序列（**escape sequence**）
New-line	\n
Horizontal tab	\t
Vertical tab	\v
Backspace	\b
Carriage return	\r
Form feed	\f
Backslash	\\
Double quote	\”
ASCII code ddd（octal）	\ddd

3

建立 SKILL 函式

1.基本概念

在第一意之中吾人已大致介紹函式的基本觀念。 在本章中則更進一步告訴使用者如何去定義一個函式，以及全域或區域變數。

事實上，在 SKILL 裡面提供了不同型式的函式，分別是 *lambda*，*nlambda*，及 *macro* 三類。SKILL 以不同的方式來處理這三種函式：

- 大部份我們所定義的函式屬於 lambda 函式。 SKILL 會先將參數的值賦與函式的形式參數（formal parameter），然後再計算函式的值。
- 有一些 SKILL 的內建函式是所謂的 *nlambda* 函式，此種函式只有單一的形式引數（formal argument）。當呼叫一個 *nlambda* 型的函式時，SKILL 會將所有傳入的真實參數集合成一個串列。然後指定給唯一的形式引數。
- macro 函式與 lambda 函式的文法雷同，但內涵則大不相同。 Macro 的計算是在編譯時期進行的，而非執行時期。

2.文法函式

SKILL 提供一組文法函式，使用者可以利用這些函式來定義新的函式。大部份時候你應該使用其中的 procedure 或 defun 函式。

2.1 procedure

使用 procedure 來定義函式大概是最普遍的方式，任何用其他方式定義的函式都可以用 procedure 的函式來定義之。下面是一個例子：

```
procedure( trAdd(x y)
    “Display a message and return the sum of x and y”
    printf(”Adding %d and %d……%d \n”x y x+y)
    x+y
    ）    ⇨  trAdd
trAdd(6 7)        ⇨  13
```

2.2 lambda

lambda 函式可以定義一個沒有名稱的函式。它的回傳值就是一個函式的物件，可以被指定給一個變數來儲存。例如：

```
trAddWithMessageFun = lambda(（x y）
    printf(”Adding %d and %d …%d \n” x y x+y)
    x+y
    )    ⇨  funbj: 0x1814b90
```

而已宣告之函式物件可以傳遞給 apply 函式來執行：

```
apply(trAddWithMessageFun ’(trAddWithMessageFun ’(4 5））  ⇨ 9
```

通常吾人不太會用到 lambda 函式的宣告，只要用 procedure 函式即可。

2.3 其他文法函式

除了上述兩個文法函式之外，尚有幾個特殊之文法函式。 n*procedure* 函式是針對舊的程式版本要升級時，爲了相容性考慮才會用到的， 在新完成的程式中不應使用到。此一函式允許你的函式用 procedure 宣告加上@rest 選項的方式， 來接收暫訂數目的引數。 也允許使用者用 *defmacro* 函式來接收未代入值之引數。

至於 *defmacro* 函式則提供了一個定義 *macro* 函式的方法。 你可以用 macro 來定義自己風格的 SKILL 語法，而 macro 則負責在編譯時期將你自訂的文法轉成 SKILL 的普通敘述，以供後續之編譯與執行。

mprocedure 則是 defmacro 另一個更基本的替代選擇。 *mprocedure* 只有單一個引數，整個使用者自己的語法型式是完整不動地傳給 *mprocedure* 函式的。 不要用此一函式在新寫的程式中，它主要的目的是供修改舊版的

程式中的函式，以與舊版的系統相容。 在新寫的程式中請用 *defmacro* 來定義函式，如果你必需去接收一些未定數目的、未代入值的引數的話，可以使用*@rest* 參數。

2.4 綜合整理

下表是針對文法函式的內容作一整理：

文法函式	函式型態	引數計算（**evaluation**）	執行
Procedure	lambda	實際引數的值被計算出來，並且傳給對應之形式引數(formal arguments)	在函式中的表示式是在執行時求出其值，並回傳最後一個敘述的值
Defmacro	macro	實際引數的值沒有被計算出來，直接將表示式傳給對應之形式引數	在函式本體內的每個表示式在編譯時期會先被展開，而最後的結果才會被編譯
Mprocedure	macro	整個函式呼叫都被對應到單一個形式引數	在函式本體內的每個表示式在編譯時期會先被展開，而最後的結果才會被編譯
Nprocedure	nlambda	所有的實際引數都不會被展開計算，並被集合成一個串列，再對應給單一個形式引數	在函式中的表示式是在執行時求出其值，並回傳最後一個敘述的值

3.定義參數

使用者可以透過在形式引數之中加入一些@選項來決定實際引數要如何被傳給形式引數。 @的選項主要有三個，*@reset*、*@optional*、*@key*。

3.1 @reset 選項

使用@reset 選項可讓使用者在呼叫函式時可以傳遞任意數目之參數(存在一個串列之中) 給這個函數。 下面的例子顯示使用@reset 的好處：

```
procedure（trTrace（fun @rest args）
    let（（result）
        printf（”\nCalling %s passing %L” fun args）
```

```
        result = apply（fun args）
        printf（"\nReturning from %s with %L\n" fun result）
        result
        ）; let
    ）; procedure
```

如果呼叫

```
trTrace（'plus 1 2 3）  ⇨  6
```

結果在 CIW 上顯示

```
Calling plus passing（1 2 3）
Returning from plus with 6
```

傳遞給 trTrace 的參數個數在不同的呼叫中皆可不同。

3.2 @optional 選項

@optional 提供使用者另一種可指定不同參數個數的函式使用方法。 同時，使用者也可以對每一個參數指定一個內定值， 使得當呼叫函式未指定某一參數的值時，SKILL 可以指定內定值給此一參數； 如果沒有預設參數的內定值，則其內定值自動設爲 *nil*。

如果使者在 procedure 的引數列定義裡面放了 *@optional* ，則任何在其後的參數都是“可給值，可不給值”的。以下是範例程式：

```
procedure( creatBBox( w h  @optional( x 0) (y 0))
    "Return a bounding box with lower left @  x:y"
        list(  x:y          ; lower left point
               x+w:y+h）   ; upper right point
    ）;  procedure
```

在上例中，當呼叫函式時引數 *h* 與 *w* 必須傳值給它，但是 *x* 與 *y* 是選擇性的參數，可傳值也可不傳值給它。 如果沒有傳的話，則 *x* 或 *y* 便會用內定的值，也就是 0。以下是一些執行的結果：

```
createBBox（3 5）          ⇨（（0 0）（3 5））
createBBox（3 5 7）        ⇨（（7 0）（10 5））
createBBox（3 5 7 9）      ⇨（（7 9）（10 14））
```

3.3 @key 選項

@optional 是依照函式的引數順序來判斷那些實際參數指定給那些形式引數。若使用 *@key* 的選擇項則可以讓使用者依不同的順序來決定那些引數要傳值給它，那些不要。以下是一個例子：

```
procedure（createBBox（@key（w 0）（h 0）（x 0）（y 0））
    “Return a bounding box with lower left @ x:y ”
    list（  x:y          ; lower left point
          x+w:y+h        ; upper right point
    ）  ;  procedure
```

```
createBBox（）                ⇨（（0 0）（0 0））
createBBox（?h 10）           ⇨（（0 0）（0 10））
createBBox（?w 5 ?x 10）      ⇨（（10 0）（15 0））
```

注意的是，*@optional* 與*@key* 是互斥的，它們不能同時出現在一引數列。

4.型態檢查

不像一般傳統的程式語言是在編譯時期做型態檢查的動作，SKILL 是在函式被執行時才做動態的型態檢查。 每一個 SKILL 的 *lambda* 或 *macro* 函式都可以在宣告引數的串列中加入一個“引數樣板”（argument template），以定義所要求的引數資料型態。 但 *mprocedure* 函式並沒有此種功能。

在引數樣板使用的字元代表不同的資料型態，這在第二章中有提到，除基本的之外，使用者也以用代表“組合”型態的字元符號，來代表兩種以上的型態。表列如下：

字元	代表意義
S	符號或字串
N	數值，包括定點及浮點數
U	函式－ 不管是函式的名稱，或是 lambda 函式的本體（list）
G	任何資料型態

以下是例子：

```
procedure（f（x y “nn”）  x*x + y*y）
```

此時 “nn”代表函式 *f* 接受兩個數值的引數。

5.註解函式

SKILL 提供一種方法來宣告一個註解，當你在建立一函式的時候，做法如下例：

```
Procedure（Plus（x y）
    “Returns the sum of x and y”
    x+y
    ）
```

其中在引數宣告之後的第一行字串即爲註解。

6.區域及全域變數

6.1 定義區域變數

SKILL 提供一個 *let* 函式來建立一暫存值給區域變數。 如下例：

```
Procedure（trGetBBoxeight（bBox）
    “Returns the height of the bounding box”
    let（（ll ur lly ury）
        ll    = car（bBox）
        lly   = cadr（ll）
        ur    = cadr（bBox）
        ury   = cadr（ur）
        ury – lly
    ）; let
）; procedure
```

其中 *ll* ，*ur*，*lly*，*ury* 是宣告的區域變數，其初值是 *nil* 。使用 let 函式可以啓始一個非 nil 值的區域變數。下面是一個例子：

```
procedure（trGetBBoxHeight（bBox）
    “Returns the height of the bounding box”
    let（（（ll car（bBox））（（ur cadr（bBox））lly ury）
        lly   = cadr（ll）
        ury   =cadr（ur）
        ury – lly
```

）; procedure

另外，使用 *prog* 函式也可以宣告區域變數，其語法如下：

prog（(local variables）your SKILL statements）

6.2 測試全域變數

應用程式通常會啓始化一或多個全域變數。當一個應用程式第一次執行時，其全域變數通常是 *unbound* 的，在此情況試圖去使用全域變數傳回值會造成錯誤。

吾人可以使用 *boundp* 函式來檢查一個變數是否是 *unbound*。舉例如下：

boundp（'Items）&& Items

當 *Items* 是 *unbound* 時，上式傳回 *nil*；否則，傳回 *Items* 的值。

6.3 重新定義既有之函式

當進行偵錯的動作時，使用者常常需要重新定義一個函式的內容。一般 procedure 定義的建構方式允許使用者去重新定義既有之函式，但前提是函式不得是已啓動“防寫保護”的狀態。要啓動 writeProtect 開關，必須執行下列函式：

sstatus（writeProtect nil）

除了偵錯的目的之外，擁有對同樣函式內容進行多重定義的能力，有時也是很方便的。例如，在開放式模擬系統（Open Simulation System）“內定”的網絡（netlist）函式可以被使用者自訂的函式取代，便是一例。

4

資料結構

1.存取運算子

有幾個資料存取（access）運算子有類推（generic）的特性，換言之，同樣的語法，可以用於不同資料型態資料的存取。以下是三個運算子：

■ -> 運算子

此一運算子可用於 disembodied property lists，defstruct，association table，和使用者自訂型態。用以去參考某個 property 的值。

■ ~> 運算子

此運算子是前述箭頭運算子的更廣義的定義。 當直接用在一個件時，其作用與箭頭運算子相同，但此一運算子也可以接受串列。

■ [ ] 運算子

此運算子爲陣列存取文法運算子，可用來存取陣列的元赤素，或是關聯式串列的“鍵一值”對的內容。

2.符號

在 SKILL 裡面符號（symbol）與變數是同樣的意思，經常交替使用。每個符號包括四個內涵（或是存放位置）： 顯示名稱（print name）、值（value）、函式鏈結（function binding）、屬性串列（property list）。 除了名稱之外，其他項目都是可有可無，而且通常不建議同時給一個符號值和函式鏈結。

2.1 產生符號

當在程式中第一次寫到一個符號時，系統便自動建立此一符號。 當系統建立一個新的符號時，其值是設爲 *unbound*。 所一般情況使用者不用做明顯的建立符號的動作，不過 SKILL 仍提了一些符號建立的相關函式。

例如，使用者可以用下列函式產生新的符號

gensym（'a） ⇨ a1

gensym（'a） ⇨ a2

給定一個基本名稱 *gensym* 函會產生一新的符號，同時爲了避免重複，系統會自動判斷在基本名稱之後加上索引數值，來當做新符號的全名。 所產生的新符號對應的值是 *unbound*。

另外，也可以用 *concat* 函式來產生新的符號，如果使用者想連合幾字串來產生新的符號名稱的話。

2.2 符號的顯示名稱

符號名稱可以包括文數字（*a~z*，*A~Z*，0~9），底線（_），問號（？）。如果名稱的第一字元是數字，則其前須放置一個倒斜線（＼）字元。 事實上，非前述的字元也可用在符號名稱中，但是每一個字元的前面要加上一個倒斜線。

吾人可以使用 *get_pname* 函式來取得符號的名稱。此一函式看似多餘，但當程式的處理中用到一個變數的值是“符號”時，就很管用了。例如：

A='S111

get_pname（A） ⇨ A

2.3 符號之值

符號的值可以是任何型態的資料，包括“符號”型態。吾人可以使用＝運算

子來指定一個值給一個符號。 函式 *setq* 即相當於＝運算子，因此下面兩式是相同的

A = 200

setq（A 200）

而要取得一個符號的值只要使用其名便可：

A ⇨ 200

如果要參考整個符號變數本身（不是只有值而已），則可以使用單引號運算子：

position = ‘A ⇨ A

也可以用間接的方式來指定一個值給一個符號，或是取得一個符號的值：

set（position 200）

symeval（position） ⇨ 200

SKILL 處理全域變數、區域變數和函式參數的方式和 C 語言不同，SKILL 的符號可以代表全域和區域變數，一個符號在任何時候都可以被使用，差別只是取得什麼值。SKILL 視每個符號的值放在一個堆疊中，而符號的目前值就是堆疊的最上面的項目，改變目前的值就是改變堆疊最上面元素的內容。 而當程式控制流程改變進入了 *let* 或是 *prog* 表示式的執行時，系統便會塞入一個暫存值進入該符號的值所存放的堆疊中。

3.符號之函式鏈結

當吾人宣告一個 SKILL 函式時，系統便用此函式的名稱來建立一個符號，並用以存放函式的定義。 而當我們重新定義一個函式時，同樣的名稱符號不變，只是之前存放的函式內容定義會被棄掉。

與符號的值不同的是，函式定義的資料不會因爲進入或離開 *let* 或 *prog* 表示式的執行而有所改變。

4.符號的屬性列

屬性列包括了所有的“屬性項目一值”的配對。每一個“名稱一值”對以相鄰的兩元素的方式存放在屬性列裡面，屬性名稱必須是符號名稱，屬性的值則可能是任意資料型態。

4.1 基本存取函式

每當一個符號產生時，SKILL 便會自動爲其配上一個屬性列，其起始內容設爲 *nil*。 欲設定一個符號的屬性列可以用下列函式

setlplist（'A '(x 100 y 200)） ⇨ （x 100 y 200）

要取得一個屬性列的內容可用下列函式

plist（'A） ⇨ （x 100 y 200）

要取得一個符號的某一個性質，可以使用點運算子

A.x ⇨ 100
getqq(A X) ⇨ 100
getqq(A Z) ⇨ nil

點運算子不可以巢狀方式使用，在點運算子的左右兩側必定是符號。 點運算子對應的函式是 *getqq*。 如果你試圖去取得一個不存在的性質的值，則所得到的回傳值會是 *nil*。 而如果你指定一個值給一個不存在的屬性時，則該性質會被加入屬性列之內。

使用箭頭 (->) 運算子和指定 (=) 運算子，可提供一個簡單方式來進行非直接地存取性質到一個符號的屬性列的動作。如下例：

```
temp1 = 'A
A.x = 100
A.y = 200
temp1->x                ⇨  100
temp1->y                ⇨  100
temp1->x = 150          ⇨  150
putpropq（temp1 150 x）⇨  150
```

putpropq 函式的作用相當於箭頭運算子加上指定運算子，上例的最後兩敘述的作用相同。

4.2 重要考慮

基本上，每一個符號對應的性質列都是全域性的， 無論何時當你傳遞一個符號給一個函式，該函式便可以異動符號的性質列的內容。 考慮以下的例子：

```
A = 1
let（A）
    A = 0
    A.x = 5
    ）;  let
x           ⇨  1
plist（'x）⇨（example 5）
```

當程式執行離開了 let 算式之後， x 的值回復爲原來的值，但是屬性列的內容仍然保留改變之後的結果。

5.不具本體之屬性串列

一個無本體之屬性列（disembodied property list）在邏輯上可看成類似 C 語言裡面的結構（structure），但不同於 C 的是，在不具本體之屬性列中，我們可以動態地增加或移除欄位。而箭頭運算子也可以用來存取串列中的項目。

在底下的例子中，一個無本體屬性列被用來表示複數。

```
procedure（createComplex（@key（real 0）（imag 0））
    let（（result）
        result = ncons（nil）
        result ->real = real
        result ->imag =imag
        result
    ）  ; let
）  ; procedure
complex1 = createComplex（?real 3 ?imag 7）      ⇨（nil imag 7 real 3）
complex2 = createComplex（?real 2 ?imag 9）      ⇨（nil imag 9 real 2）
i = createComplex（?imag 1）                     ⇨（nil imag 1 real 0）
procedure（createAddition（c1 c2）
    createAddition（
        ?real     c1 ->real + c2 ->real
        ?imag     c1 ->imag + c2 ->imag
    ）
```

```
）; procedure
procedure（createMultiply（c1　c2）
    createComplex（
        ?real     c1 ->real * c2 ->real – c1->imag * c2->imag
        ?imag     c1 ->imag * c2->real + c1 ->real * c2 ->imag
    ）
）; procedure
createMultiply（i i）      ⇨ （nil　imag　0　real　-1）
```

在某情況之下使用無本體屬性列有其好處。 例如有時候你想建立一個屬性列但沒有適當的符號來做爲依附的本體。 有時候如果你爲每個記錄動作都建立一個符號，有時可能會耗掉大量的符號，SKILL 在管理變數上會更沒有效率。另一個原因是傳遞無本體屬性串列的參數較傳遞符號容易。

值得一提的是，你可以將無本體屬性串列當成值指定給一個符號，但這不會影響此一符號原有的屬性列。

5.1 重點考量

在指定一個無本體屬性串列給一個變數時， 要特別注意一件事。我們舉下例說明：

```
comp1 = comp2
comp1 = = comp2          ⇨　t
eq（comp1 comp2）        ⇨　t
```

由此可見 comp1 與 comp2 的內容完全相同，事實上，SKILL 是以指標的方式將二者都指向同一記憶體位址，如果用箭頭運算子去修改 comp1 的 real 屬性項目，則 comp2 的 real 屬性項目也會同步改變。要避免此種情況，須用 copy 函式：

```
comp1 = copy（comp2）
comp1 = = comp2          ⇨　t
eq（comp1 comp2）        ⇨　nil
```

5.2 其他屬性串列函式

要增加一個新的性質到一個符號的屬性列，或是一個無本體的屬性串列之中，可以使用 *putprop* 函式。 如果欲加入的性質項目早已存在，則 *putprop* 函

式會將該項目的舊值換成新值。 *putprop* 函式是一種 *lambda* 函式，換言之，呼叫此一函式時所有的實際引數項目都會先計算完後再代入形式引數。 而如果要取得一個有命名的屬性串列的某一屬性的值的話，要使用 *get* 函式。 *get* 的作用與 *putprop* 剛好相反。

舉例如下：

putprop（'U20 3+3 'pins） ⇨ 6
U20.pins = 3 + 3 ⇨ 6
get（'U20 'pins） ⇨ 6
U20.pins ⇨ 6

第一跟第二行敘述的作用是相同的。

要增加一些屬性給一個符號或是無本體屬性串列， 而又不想將函式的參數先行計算出來的話， 可以呼叫 *defprop* 函式。 舉例如下：

defprop（a 2 x） ⇨ 2
defprop（a 2 * 3 x） ⇨ 2 * 3

要除去屬性列中的一個屬性可用 remprop 函式，舉例如下：

setplist（'U2 '(x 200 y 300)） ⇨ (x 200 y 300)
putprop（'U2 8 'pins） ⇨ 8
plist ('U2) ⇨ （pins 8 x 200 y 300）
get ('U2 pins) ⇨ 8
remprop ('U2 'x)
plist（'U2） ⇨ （pins 8 y 300）

6.字串

字串可視爲一維的字陣列，在本節中介紹 SKILL 提供常用之字串處理的函式。

6.1 字串串接

用 *buildString* 函式可以由一串字串來結合成一個大字串，在大字串用指定的分隔字元來連接原來的各個字串。 如下例：

buildString（’（”word” ”doc”）”.”） ⇨ “word.doc”
buildString（’（”come” ”go”）”/”） ⇨ “come/go”
buildString（’（”x” “y”）” “） ⇨ “x y”
buildString（’（”x” “y”）”“） ⇨ “xy”

由上面可知道 buildString 的最後一個引數是指定的分隔字元。

要串接幾個字串來產生一個新的字串可以使用 strcat 函式，原有的字串不會受到影響：

strcat（”How” “are” “you”） ⇨”How are you”

要取第二個字串中的前 n 個字元，再附加到第一個字串之後來產生一個新的字串，可以使用 *strncat* 函式。原有的字串不會受到影響：

strncat（”abc” “defgh” 4） ⇨ “abcdefg”

6.2 字串比較

在此介紹三個字串比較的函式。 首先， 如果要依“文字順序”比較兩個字串或者是符號的大小，可以使用 *alphalessp* 函式。 如果第一個引數比第二個引數小，則回傳值是 t。而 *alphalessp* 常常搭配 *sort* 函式使用來排序一列字串。使用如下：

```
str = ‘（”cfg” “abd” “abc”）
sort（str ‘alphalessp） ⇨ （”abc” “abd” “cfg”）
```

單純要比較兩個字串的大小可以用 strcmp 函式。 舉例如下：

strcmp（”ab” “aa”） ⇨ 1
strcmp（”ab” “ab”） ⇨ 0
strcmp（”ab” “ac”） ⇨ -1

另一個函式是 alphaNumCmp，可提供比較兩個字串或符號的大小，比較的方式可以是依“文字順序”或是“數值大小”。如果第三個引數是 non-nil 的，而且前兩個引數是表示純數值的字串，則進行數值大小之比較。舉例如下：

alphaNumCmp（”x” “y”） ⇨ -1
alphaNumCmp（”y” “x”） ⇨ 1
alphaNumCmp（”x” “x”） ⇨ 0
alphaNumCmp（”10” “12” t） ⇨ 0

如果只要比較兩個字串的前面 *n* 個字元，則可以用 *strncmp* 函式，其比較結果可以看回傳值，表示的方式與 *strcmp* 相同：

strncmp（”abc” abd” 2） ⇨ 1
strncmp（”abc” abd” 2） ⇨ -1

6.3 處理字串中之字元

如果要得知一個字串的長度，可以使用 *strlen* 函式：

strlen（”xyz”） ⇨ 3
strlen（”\001”） ⇨ 1

如果要取得字串中的第幾個字元，可以使用 *getchar* 函式：

getchar（”xyzw” 1） ⇨ x
getchar（”xyzw” 5） ⇨ nil

注意，getchar 回傳的是取字元爲名的“符號”，不是“字串”。

要取得字串 *str1* 中第一次出現字串 *str2* 開始的部份，可以使用 *index* 函式。如果是要取得 *str2* 最後一次出現到最後的部份，可以使用 *rindex* 函式。

index（”xyzwxyzw” “y”） ⇨ “yzwxy
index（”xyzwzyzw” “wz”） ⇨ “wzyzw”
index（”xyzwzyzw” “ab”） ⇨ nil
rindex（”xyzwzyzw” “yz”） ⇨ “yzw”

要取得一個字元在符號或是字串中的位置，可以使用 nindex 函式。

nindex（”abcd” ‘c） ⇨ 3
nindex（”abcdef” “cde”） ⇨ 3
nindex（”abcdef” “xyz”） ⇨ nil

6.4 建立子字串

要拷具一個字串的一部份成爲一個新的字串要用 *substring* 函式。如下例：

substring（”abcdefg” 3 , 4） ⇨ “cd”
substring（”abcdefg” 4 , 3） ⇨ “def”

其中第二個引數代表開始取子字串的位置，第三個引數代表要取幾個字元。

parseString 函式的作用是根據指定的分割字元， 將一個字串分解成若干個子字串。用法如下：

parseString（”How are you”） ⇨（”How” “are” “you”）
parseString（”proposal” “o”） ⇨（”pr” “p” “sal”）
parseString（”How are you? Fine” “?”） ⇨（”How” “are” “you” “Fine”）

第二個引數是若有指定，則 SKILL 會用此引數當作切割字元來分斷原字串。若沒有給第二個引數，則以空白字元做爲分斷字元。

6.5 大小寫轉換

要將字串中的小寫字換成大寫字可使用 upperCase 函式；反之，則使用 lowerCase 函式：

upperCase（”Hello !”） ⇨ “HELLO !”
sym = “Hi” ⇨ “Hi”
upperCase（sym） ⇨ “HI”
lowerCase（”Hello !”） ⇨ “hello !”

7. Defstructs

Defstructs 是幾個變數的集合，這些變數的型態可以都不一樣，但在同樣的一個結構（structure）的名稱之下被處理。 這種資料結構類似於 C 語言中的 *struct*。以下是一個 *defstruct* 的例子：

defstruct（s_name s_slot1 s_slot2） ⇨ t

一旦一個結構被建立起來，其行爲模式與無本體屬性串列雷同，但儲存方式較有效率且存取時較短。 結構可以動態地增加新的項目，不過那些動態產生的項目存取的效率和空間的使用率不若靜態宣告者。

假設 *struct* 是一個結構的例子，則

struct->slot

的意義是傳回 *slot* 這個欄位對應的值；

struct->slot = newval

的意義是指定新的值給 *struct* 的 *slot* 欄位；

struct->?

會傳回 *struct* 的所有欄位名稱；

strcut->??

傳回 *struct* 的屬性串列。

7.1 其他 defstruct 函式

要測試一個 SKILL 的物件是否是某種結構的例子，可以使用 defstructp 函式。 如果回傳值是 t ，表示答案是肯定的；否則回傳值爲 nil 。以下是一個例子：

defstructp（x structA） ⇨ t

defstructp（x “structA”） ⇨ t

第二個引數是結構名稱，可以是符號，也可以是字串型式。

顯示一個結構的內容可以用 *printstruct* 函式。即使此一結構中包含有一個欄位也是結構型態，此一函式也能遞迴地印出所有的欄位資料。

要拷貝一個結構的內容給另一個新的結構可以使用 *copy_<name>* 函式，此一函式是當 *defstruct* 函式在建立一個結構時產生的。 如果要遞迴地拷貝一個結構的內容，則要使用 *copyDefstructDeep*。

8 陣列

在 SKILL 中使用陣列必須先行宣告，這跟使用變數不同。SKILL 的陣列有幾個特點：

- 陣列中的元素可以分屬不同型態
- SKILL 提供執行時期的陣列範圍檢查
- 陣列本身是一維的，但你可以建立陣列的陣列，如此相當於二維的陣列，以此類推，你可以建立更高維的陣列。
- 陣列範圍檢查是在執行時期，每一次有人使到陣列時就會進行的動作

8.1 配置指定大小空間給陣列

使用 declare 函式可以配置指定大小的空間給一個陣列。 如下例：

```
declare( week[7])          ⇨  array[7]:9780700
week                       ⇨  array[7]:9780700
type(week)                 ⇨  array
arrayp（week）             ⇨  t
days = '( Mon  Tue  Wed  Thu  Fri  Sat  Sun)
for（day 0 length( week )-1
     week[day] = nth(day days)）
```

上述的 type 函式的作用是傳回變數的型態。 要注意陣列的元素編號是由 0 開始的。

當一個陣列的名稱出現在一個指定敘述（＝號敘述）的左邊， 而並未帶有索引時，只有陣列物件本身被指定給別的變數， 陣列內所存的值並沒有做拷貝的動作。 因此有可能給同一個陣列不同的名稱。 這種情形有點像在 C 語言中用一個指標 *b* 去等於一個陣列 *a* ，則不管用 *a* 或 *b* 都可以去存取同一個陣列的內容。

下面是一些應用和說明：

```
declare（a[5]）      ；宣告含有五個元素的 a 陣列
b = a                ；b 指向 a 陣列
declare（c[4]）      ；宣告含有四個元素的 a 陣列
c[0] = b             ；c[0]指向 a 和 b 共同指的陣列位置
c[0][2]              ； 讀取 a 陣列的第 3 個元素
```

9.關聯式表格

關聯式表格（association tables）是一堆“鍵／值”（key/value）對的集合。可以被用做鍵者有整數、浮點數、字串、串列、符號，和些許使用者定義的資料型態。使用關聯式的表格，其資料找尋的效率和方便性較使用無本體屬性串列、陣列等要高。

9.1 初步化表格

使用 makeTable 函式可以定義並初始化一個關聯式表格。 這個函式接受的第一個引數是一個字串，做爲列印用的表格名稱，第二個引數則是可有可無的，如果有的話代表內定值，每當讀取表格時所使用的鍵值不存在表格之中時，此一內定值便會被傳回。考慮下例：

```
table1 = makeTable("table1" 0)        ⇨  table:table1
tablep(table1)                        ⇨  t
table1[1] = "blue"                    ⇨  "blue"
table1["two"] = '( r e d )            ⇨  (r e d)
table1[3]                             ⇨  0
length(table1)                        ⇨  2
```

其中 *tablep* 函式驗証 table1 是否爲一個表格； *length* 函式則是算表格內有多少鍵值。

9.2 處理表格

SKILL 提供了一些處理關聯式表格的函式。分別介紹如下：

首先，吾人可以使用 *tablep* 函式來檢查某一個資料項是否爲表格：

```
table1 = makeTable("table1" 0)        ⇨  table:table1
tablep(table1)                        ⇨  t
tablep(5)                             ⇨  nil
```

如果要將關聯表格的內容轉換成關聯串列，則使用 *tableToList* 這個函式。此一轉換對資料處理的效率而言，並不有利，因此最好別做這樣的轉換；相反地，只要利用它來逐步地查看表格的內容即可。

```
tableToList( table1)
    ⇨   (("two"(r e d)(1 "blue"))
    ⇨
```

要將關聯表格的內容寫到檔案去要用 *writeTable* 函式；要將一檔案內容讀入附加到一個關聯表格之內，要用 *readTable* 函式。舉例如下：

```
writeTable("out1.log"  table1)      ⇨  t
readTable("out1.log"  table)        ⇨  t
```

要將關聯表格的內容印出成表格形式，則可使用 printstruct 函式。此一函式會遞迴地印出整個表格的內容。

```
printstruct（table1）
    ⇨  1 :  “blue”
       “two” : （r e d）
```

9.3 瀏覽表格

使用者可以使用 *foreach* 函式來查詢表格中的每一個鍵值， 使用此一函式的結果是印出表格中所有的“鍵／值”對。另一個類似的函式是 *forall*，如果在表格中的每個鍵都是字串，值都是整數的話，便可以使用 *forall* 函式。

```
foreach（key table1
        println（ list（key table1[key] ））
    ）
forall（key table1
        stringp（key）&& fixp（table1[key]）
    ）
exists（key table1
        stringp（key）&& fixp（table1[key]）
    ）
```

上述 *exists* 函式的作用是，檢查是否 *key* 是存在 *table*1 之內的鍵值，若有的話就執行後面的敘述。

10. 關聯串列

所謂關聯式串列（association list）其實就是“鍵／值”對的串列。一個關聯串列是一個串列的串列。舉例如下：

```
assocList = ‘（  （”a” 1）（”b” 2）（”c” 3））
```

吾人可以用 *assoc* 函式來讀取一個關聯串列的項目：

```
assoc（”b”  assocList）      ⇨  （”b” 2）
```

assoc（”c” assocList） ⇨ （”c” 3）

吾人也可以用 rplaca 函式來更關聯串列的項目：

rplaca（cdr（assoc（”b” assocList））”two”） ⇨ （”b” “two”）

assocList ⇨（（”a” 1）（”b” “two”）（”c” 3））

5

算術與邏輯表示式

1. 計算表示式的值

運算表示式（expression）是 SKILL 的物件，其計算結果也是 SKILL 的物件。 SKILL 執行計算的過程相當於一連串的函式求值運算過程。事實上，SKILL 的程式就是由一堆表示式來構成，用以解決特定問題的。

在 SKILL 中的表示式可分成三種基本類別： 常數（constant），變數（variable），以及函式呼叫（function call）。

常數的值的計算很簡單，就是它本身！ 如 146， “ab”，2.4 等都是常數。變數的值是在編譯時期被儲存的，而在執行時期時回傳，例如

```
a, b, vars
```

等等都是變數。如果在執行時遇到要計算未設值的變數的算式的時候，SKILL 會傳回一個錯誤訊息，告訴使用者所用的是 *unbound* 變數。要注意的是：*unbound* 跟 *nil* 是不同的，unbound 代表的是“根本沒有任何東西” ！

至於函式的呼叫則是透過呼叫函式名稱，加上一串參數列來完成。例如

```
f(a b), abs(-30), exit( )
```

2. 建立算術與邏輯之表示式

2.1 用單引號避免計算

通常你可能只想引用一個表示式，但不想去計算其值，此時可以在該表示式之前加上一個單引號。例如

a = 10 ⇨ 10

‘a ⇨ a

a ⇨ 10

可以發現當吾人加上單引號於變數 *a* 之前時，回傳值是變數的名稱，而不是其值了。

在使用一個 list 時更要小心，通常要在 list 之前加上單引號，才不會錯誤。這是因爲在 SKILL 裡面一般的括弧（）表示式可用來表示函式或資料，如果輸入了（*f x y*），則解譯器會視 f 爲函式名稱，而不是一個普通的資料 list。

2.2 算術與邏輯運算子

所有的算術或邏輯運算子都會被轉換成對應的 SKILL 函式，以便執行對應的功能。這些運算子都列在下列的表格之中。

資料存取運算子	用法	函式名稱
[ ]	a[index]	arrayref
	a[index] = expr	setarray
<>	x<bit>, x<msb:sb>	bitfield1, bitfield
<:>	x<bit>=expr	setqbitfield1
	x<msb:lsb>=expr	setqbitfield
‘	‘expr	Quote
.	g.s , g->s	getqq, getq
->	g.s = expr, g->s = expr	putpropqq, putpropq
~>	d~>s, d~>s = expr	

單運算元運算子	用法	函式名稱
++	++a, a++	preincrement, postincrement
--	--b, b--	predecrement, postdecrement
-	-n	minus
!	!expr	null
~	~x	bnot

雙運算元運算子	用法	函式名稱
**	a1 ** a2	expr
*	a1 * a2	times
/	a1 / a2	quotient
+	a1 + a2	plus
-	a1 – a2	difference
<<	a1 << a2	leftshift
>>	a1 >> a2	rightshift
<	a1 < a2	lesssp
>	a1 > a2	greaterp
<=	a1 <= a2	leqp
>=	a1 >= a2	geqp
= =	g1 = = g2	equal
!=	g1 != g2	nequal
&	a1 & a2	band
~&	a1 ~& a2	bnand
^	a1 ^ a2	bxor
~ ^	a1 ~^ a2	bnxor

雙運算元運算子	用法	函式名稱
|	a1 a2	bor
~	a1 ~ a2	bnor
&&	left && right	and
||	left || right	or
:	a1 : a2	range
=	a = expr	setq

以下則是針對某些運算子做進一步介紹

更多的算術運算子

算術運算子	命令
前增加運算子（++使用在變數名稱之前）	將一變數名稱的引數的值增加 1（此值必須是數字）。並將值存到變數中，然後再增加其值。
後增加運算子（++使用在變數名稱之後）	將一變數名稱的引數的值增加 1（此值必須是數字）。並將值存到變數中，然後再增加其值。然而，它將初始值回傳儲存在變數中，並呼叫函數的結果。
前減少與後減少運算子	相似於前增加與後增加運算子，但是它們反而是將其數值減少 1 而不是加 1。
階層運算子（：）	計算引數和回傳值結果兩者作爲兩元素的 list，一對資料值群組提供了一非常方便的方法，舉例說明如 1:3 回傳的 list 爲（1 3）。

2.3 預先建好之算術函式

除了基本的中序算術運算子之外，還有一些預先定義好的函式。

預先定義好之算術函式

對照表	結果
一般函式	
add1（n）	n+1
sub1（n）	n-1
abs（n）	n 的絕對值
exp（n）	e 的 n 次方
log（n）	n 的自然對數
max（n1 n2…）	從 n1 n2 的引數中取最大值
min（n1 n2…）	從 n1 n2 的引數中取最小值
mod（x1 x2）	*x2* 除以 *x1* 取其整數餘數
round（n）	將 n 取成整數
sqrt（n）	將 n 取平方根
sxtd（x w）	將 *x* 向右記號延長 w 位元
zxtd（x w）	將 *x* 向右零值延長 w 位元

算術函式的定義

對照表	結果
三角函式	
sin（n）	sin
cos（n）	cosine
tan（n）	tangent
asin（n）	arc sine
acos（n）	arc cosine
atan（n）	arc tangent
亂數產生器	
random（x）	傳回一個介於 0 到 x-1 之間的亂數。若沒給引數，則傳回一個整數，其所有位元都是隨機賦予的。
srandom（x）	將亂數產生器的初始狀態設定爲 x。

2.4 逐位元邏輯運算子

共有 *bnot*、*band*、*bnand*、*bxor*、*bxnor*、*bor* 和 *bnor* 七種運算子，它們是建立在所有整數引數之上來執行所有的邏輯運算。

Bitwise 邏輯運算子

運 算 元	意 指
&	逐位元 AND
|	逐位元 inclusive OR
^	逐位元 exclusive OR
>>	左移
<<	右移
~	1 的補數

2.5 位元欄位運算子

Bit field 運算子專門處理儲存在字組中的 32 位元。是爲了避免與位元的命名混淆，SKILL 指定位元 0 是整數的最不重要位元（least significant bit）。

- 你可以位元欄的最左邊或最右邊的位元來指名選用一個位元欄。 或者如果欄位寬度是 1 時可以直接指名該位元來選擇欄位。
- 你可以使用整數常數或是整數變數來指明位元位置， 但是不可以使用運算表示式。

■ 所有的位元欄位皆視爲無號整數

■ 使用 sxtd 函式來對位元欄做符號橫展的動作

位元欄位的例子

X = 0b01101 ⇨ 13

在二進制中指定 x 爲 13

X<0> ⇨ 1

x 的最右邊位元等於 1

X<1> ⇨ 0

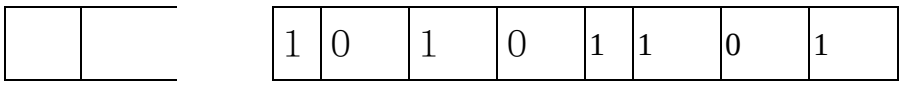
x 的位元 1 等於 0

X<2 : 0> ⇨ 5

將 x 的最右邊的三個位元值萃取出來，其值爲 101 或是十進位的 5

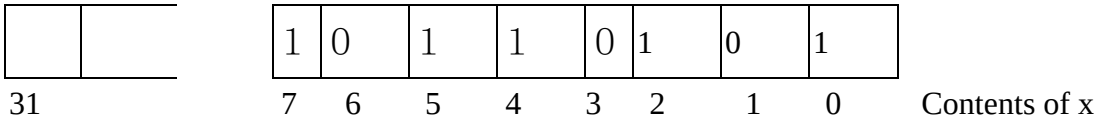
X<7 : 4> = 1010 ⇨ 173

將位元值儲存到位元 4 至位元 7 的位置



（X + 4） <4 : 3> ⇨ 2

將 X 加上 4，然後從第三及第四位元取出數值，將 173 加上 4 爲 177，SKILL 將回傳最後一個式子的結果，爲二進制的 10 或十進制的 2。



呼叫位元欄位函式的協定

因爲在文法的限制中，只有整數常數或是變數名稱才能被放在<>括弧中。要使用一個運算表式的值來指定位元的位置， 你必須從它們兩者之中指定方程式的值或是直接的呼叫位元欄位函式，但並不能使用小於（<），冒號（：），和大於（>）的中序運算子，這些位元欄位函式的呼叫協定就如下列所示：

```
bitfield1（
   x_value
   x_bitPosition ）

setqbitfiekd1（
   s_name
   x_newvalue
   x_bitPosition ）

bitfield（
   x_value
   x_leftmosBit
   x_rightmosBit ）

setqbitfield（
   s_name
   x_newvalue
   x_leftmostBit
   x_rightmosBit ）
```

2.6 混合模式的算術運算

SKILL 在處理整數與浮點數之間的差異如下：

- 假如在算術運算中所有的引數都是整數，則使用整數運算。
- 假如所有的引數都是浮點數，則使用浮點數運算。
- 當整數與浮點數混合時，SKILL 會試著保持整數運算方式直到遇到浮點數，SKILL 會切換浮點運算和傳回浮點運算的結果。

有關整數與浮點數的除法，是須要特別注意的，因爲以下兩點：

- 整數除法可縮短它的結果
- 浮點除法可計算的數值

事實上，兩數只有當是相同的型態（指的是整數與浮點數）且數值相等時才叫做相等。因此在比較整數與浮點數時，勢必要對其一進行型態轉換才能進行有意義的比較。

型態轉換函式（定點和浮點）

fix 函式將浮點數轉換成整數，而 *float* 函式是將整數轉換成浮點數，假若引數已是想要的型態，則 fix 或 float 函式會直接將引數回傳。

比較浮點數

警告

除非有兩個浮點數是被指定成完全相同的常數，或是經宙完全相同計算來產生值，否則當比較二者的值不太可能一樣。

兩個浮點數當它們顯示出來時值一樣時，其實未必是相同的， 因爲有可能二者的最小的幾個位元並不相同，而這些不會顯示出來。這種差異益發複雜， 因爲 SKILL 用倍精確的方式儲存所有的浮點數，在單一精確之下只有一些應用儲

存浮點數，當浮點數是被儲存或是恢復時，所產生的損失就是精確性降低。以下舉例說明：

```
if（ （a = = b） println（”same”））
```

這是用浮點數來做很少成功的條件式。

```
if（ （abs（a - b）/ a <  1e-6） println（”same”））
```

以範圍來比較的方式較強有力。

2.7 函式超載

有一些應用程式（像是Analog Artist™）會將 SKILL 原有的一些算術或位元階層的函式加以過載（overload），賦予新的或修飾過的語法。例如： 在類比Artist軟體有支援複數，一般在SKILL中sqrt(-1)通常是表示一個信號錯誤；但是在類比Artist 執行時，它代表傳回一有效的複數值。

因爲算術及位元階層的運算子從文法上來看只是簡單的呼叫它們對應的函式而已，經由過載其定義， 擴展其語法使能應用於新的資料型態的物件，便可以應用原來熟悉的語法來處理新的資料物件。

再次以類比 Artist 爲例，經由適當地過載加、減、乘、除這些函式，你便可使用+、-、*、/ 這些熟悉的運算子來做包含複數資料型態的算術運算。

警告

這樣過載的方式是個別應用軟體所完成的，在 SKILL 中並未支援使用者可定義的函式過載，有關詳情，請參閱個別應用軟體之使用手冊。

2.8 純整數算術運算

除了能適用於整數和浮點數的標準的算術函式之外， SKILL 提供了數種的執行速度較標準函式快一些，但只適用於整數的算術函式。 這些函式的名稱通常是在對應的標準函式名稱之前，再加上一個 *"x"*，例如：

xdifference，*xplus*，*xquotient*， 和 *xtimes*。

當使用 *sstatus* 函式開起整數模式時，在 SKILL 編譯過程中，文法分析器會將所有使用到算術運算子的地方，改成呼叫純算術運算函式。 這樣的做法可節省少量的執行時間，所通常只有在程式需要用到大量的整術算術運算時才有用處。執行

sstatus（integermode t） ⇨ t

啓用整數模式。 而執行

status（integermode） ⇨ t

會檢查整數模式是否有啓動，如果回傳值是 *t*，則表示已經啓動。 內定的值是沒有啓動。

內部的變數可視爲典型的布林開關，因爲它只接受 *t* 值和 *nil* 兩種。考慮系統的效率和安全，以內部變數型態儲存的系統變數，其值只能被 sstatus 異動，不像其他的 SKILL 變數，你可以直接去設定其值。

2.9 真（non-nil）和僞（nil）情況

不像 C 語言是使用整數來表示真（true）、否（false）的情況，SKILL 使用特殊的、非數值的基本因子 *nil* 來表示否定情況，然而正確情況則用特殊的基本因子 *t* 或是其他非 *nil* 的東西來表示。

關係運算子

關係運算子有 *lessp*(<)，*leqp*(<=)，*greaterp*(>)，*geqp*(>=)，*equal*(= =)，*equal*(!=)等，其作用於數值的運算元，並且視判斷結回傳 *t* 或 *nil* 的值。

邏輯運算子

邏輯運算子有 *and*(&&)、*or*(||)、和 *null*(!)，與前述不同的是，這些運算是作用在非數值的引數，以判斷是否（nil）或是真（non nil）的情況的其中一個。

相同與不相同的測試

你可以使用 equal(= =)和 nequal(!=)運算子來測試非數值的基本因子（項目）是否相等與否。

- 假如它們有相同的型式與相同的值，那麼兩個基本因子是相等的。
- 假如兩個串列包含剛好完全相同的元素，那麼兩個串列是相等的。

2.10 控制計算的順序

二元運算子&&和||通常被用用控制計算值的次序。

&&運算子

&&運算會先計算的第一個引數的值，假如其結果是 *nil*，直接回傳 nil，不會計算的第二個引數；假如第一個引數爲 non-nil，則&&會繼續計算第二個引數，並且回傳其結果當做此函式的值。

||運算子

||運算會先計算的第一個引數的值，假如其結果是 *non-nil*，直接回傳其值，不必計算的第二個引數；假如第一個引數爲 *nil*，則||會繼續計算第二個引數，並且回傳其結果當做此函式的值。

2.11 算術條件測試

除了六個中序的關係運算子之外，下列數個函式也可用於測試算術結果。

算術述語函式

對 照 表	結　果
minusp（n）	假如 n 爲一個負數回傳 *t* 值，否則回傳 *nil*。
plusp（n）	假如 n 爲一個正數回傳 *t* 值，否則回傳 *nil*。
onep（n）	假如 n 等於 1 回傳 *t* 值，否則回傳 *nil*。
aerop（n）	假如 n 等於 0 回傳 *t* 值，否則回傳 *nil*。
evenp（x）	假如 x 爲一個偶數回傳 *t* 值，否則回傳 *nil*。
oddp（x）	假如 x 爲一個奇數回傳 *t* 值，否則回傳 *nil*。

3. SKILL 和 C 之間的不同

在 SKILL 中算術和邏輯的運算式，大致和 C 語言來說是相同的，只有下列的一些小小差異。

- SKILL 增加一個指數運算子，以兩個星號“**”爲表示。
- 求餘數的運算子 ”%” 已被 mod 函式取代。
- 廢除條件式運算子 “？” 和 “：” ，只利用普通的 if/then/else 的控制結構來運作。
- 在 C 語言中與指標有關之間接運算子” *”和位置運算子”&”，在 SKILL

中不支援。

- 增加了 *nand*(~&)，*nor*(~|)，*xnor*(~ ^)等的逐位元運算子及其對應之函式。
- 邏輯算式的回傳值是基本因子 *nil* 或一個 *non-nil*（通常是基本因子 *t*）的值來代邏輯上的真、否情況。

4. SKILL 的述詞

下列的述詞的功能是去測試一些情況。

基本因子函式

atom 函式的作用是檢查物件是否爲基本因子，所有的 SKILL 的物件（除了非空的串列之外）都是基本因子。特殊符號 *nil* 既是基本因子也是一種串列。

```
atom（’hi）      ⇨  t
x = ‘（a b c）
atom（a）        ⇨  nil
atom（nil）      ⇨  t
```

boundp 函式

boundp 函式檢查一個符號是否有帶值。

```
x = 5                          ; binds x to the value 5。
y = ‘unbound                   ; unbinds y
boundp（’x）       ⇨  t
boundp（’y）       ⇨  nil

y = ‘x                         ; binds y to the constant x。

boundp（y）        ⇨  t        ; returns t because y evaluates to x ,
                               ; which is bound。
```

有效率地使用述詞

有一些述詞執行起來較其他的快，例如 *eq*、*neq*、*memq*、*caseq* 函式都是較其他類似功能的 *equal*、*nequal*、*member*、*case* 函式快。

equal、*nequal*、*member*、*case* 等函式只是比較受測物件的值而已，不像 *eq*、*neq*、*memq*、*caseq* 函式是判斷兩個物件是否相同，換言之，是判斷二者所佔的記憶體位置是否相同。

eq 函式

eq 函式以檢查位址方式測試兩引數是否相等，若是則回傳 *t* 值。你可以測試在判斷符號相等時，使用 *eq* 函式較使用 = = 運算子是否更有效率。以下舉例顯示使用 *equal*（= =）和 *eq* 函式的不同。

```
list1 = ‘（1 2 3）              ⇨  （1 2 3）
list2 = ‘（1 2 3）              ⇨  （1 2 3）
list1 == list2                  ⇨  t
eq（list1 list2）               ⇨  nil

list3 = cons（0 list1）（0 1 2 3）
list4 = cons（0 list1）（0 1 2 3）
list3 = list4
eq（cdr（list3） list1）        ⇨  t

alist = ‘（a b c）              ⇨  a b c
eq（’a car（alist））           ⇨  t
```

equal 函式

equal 函式的測試

- 在虛擬計憶體中假如兩引數指的是相同的物件，則回傳 *t*。
- 假若引數是相同的型式並且它們滿足完全相等（例如 *string* 的完全相同的連續特性），則回傳 *t* 。
- 假若這些引數是 *fixnum* 和 *flonum* 的混合物，且這些數是完全相同的

（例如 1.0 和 1）。

以下的測試速度會較使用 *eq* 慢一些，但是對符號或物件的比較都可以用。

```
x = ‘ cat
equal（x ‘ cat）          ⇨  t
x = = ‘dog                ⇨  nil    ;   = = is the same as equal。
x = “world”
equal（x “world”）        ⇨  t
x = ‘（a b c）
equal（x ‘( a b c )）     ⇨  t
```

neq 函式

neq 會檢查兩個引數是否不相等，若是則會回傳 *t* 值。任兩個 SKILL 運算表示都會被測試是否指向的是相同的物件。

```
a = ‘dog
neq（a ‘dog）       ⇨  nil
neq（a ‘cat）       ⇨  t
z = ‘ ( 1 2 3 )
neq（z z）          ⇨  nil
neq（’( 1 2 3 ) z）  ⇨  t
```

nequal 函式

nequal 會檢查兩個引數是否是邏輯上相等，若不是是則回傳 *t* 值。

```
x = “cow”
nequal（x “cow”）       ⇨  nil
nequal（x “dog”）       ⇨  t
z = ‘（1 2 3）
nequal（z z）           ⇨  nil
nequal（’( 1 2 3 ) z）  ⇨  nil
```

member 和 memq 函式

這兩個函式提供對於串列的成員關係的測試，*member* 函式借用 *equal* 進行測試，而 *memq* 借用 *eq* 進行測試，所以後者是較快的。假若第一個引數出現在第二個引數的串列中的話，則這些函式回傳 *non-nil* 值。

x = ‘ c

memq（x ’(a b c d)） ⇨（c d）

memq（x ‘(a b c d)） ⇨ nil

x = “c”

member（x ‘(“a” “b” “c” “d”)） ⇨（”c” “d”）

memq（’ “(a b c d c d)） ⇨（c d c d）

memq（concat(x) ’(a b c d)） ⇨（c d）

5. 型態述詞

許多的述詞函式是可用在測試資料物件的資料型式，下標 “p” 通常加在函式的名字後以表示是這類的函式。這些函式都列於下列的表格中，g(general)可以是任何的資料型式。

屬性型式

函　數	回傳值
arrayp(g)	如是 g 是一個陣列，回傳 t，否則回傳 nil。
bcdp(g)	如是 g 是一個二元函式，回傳 t，否則回傳 nil。
dtpr(g)	如是 g 不是空串列，回傳 t，否則回傳 nil。
fixp(g)	如是 g 是一個定點數（整數），回傳 t，否則回傳 nil。
floatp(g)	如是 g 是一個浮點數，回傳 t，否則回傳 nil。
listp(g)	如是 g 是一個串列，回傳 t，否則回傳 nil。
null(g)	如是 g 是 nil，回傳 t，否則回傳 nil。

numberp(g)	如是 g 是一個數字，回傳 t，否則回傳 nil。
otherp(g)	如是 g 是一個外來資料指標，回傳 t，否則回傳 nil。
portp(g)	如是 g 是一個 I/Oport，回傳 t，否則回傳 nil。
stringp(g)	如是 g 是一個字串，回傳 t，否則回傳 nil。
symbolp(g)	如是 g 是一個符號，回傳 t，否則回傳 nil。
symstrp(g)	如是 g 是一個符號或字串，回傳 t，否則回傳 nil。
type(g)	一個符號，其名稱描述 g 的型式名稱。
typep(g)	與 type(g) 作用相同。

6

控制結構

1. 控制函式

SKILL 控制函式提供使用者許多功能讓其他語言（如 C）的使用者感到很親切熟悉。這些高階的控制函式讓 SKILL 功能更加的強。

在 SKILL 語言中控制函式也是導致無效碼的理由，提供如此多的制結構必然導致有些較其他的有效率一點，而且提供的功能有不少是重複的。這通意謂著使用者會傾向於只使用他覺得最方便的幾種控制結構來寫程式， 但這種做法不一定就是最佳的做法。

任何一個已給定參數的函式的執行，會控制對於某一表示式的計算者 ，就是一個控制函式。 計算的次序可以依據求值測試條件的結果而定 ，加上一個標準的控制建構就像是 if/while/for ，SKILL 使得你很容易去定義屬於自己的控制函式，因爲在 SKILL 中的控制函式相當於在傳統程式語言中的命令敘述（statement），在本書中有時也會使用這個名詞。

1.1 條件函式

條件判斷函式會測試一個條件，並且在條件確定時 ，執行對應用之處理工作。

有四個條件函式可用在 SKILL 的程，亦即 *if*，*when*，*unless*，跟 *cond*。這些每一個都擁有它們自己的特性，因爲這四個函式實現非常相似的工作 ，程式

設計者很容易會選用不適當的控制函式。最好是根據下列的規範來選擇條件函式：

函式名稱	使用時機
if	剛好只有兩個值 true 和 false 可以考慮。
When	當測試題爲 true 時，有些命令敘述才要執行。
Unless	有些命令敘述要執行，除非測試證實爲真時。
Cond	有不只一個的測試條件，但只有一個測試條件會爲真，其對之命令敘述會執行。

在這裡將只討論 cond 函式，對於討論 *if*，*when*，*unless* 函式在第一章中已有提到。

cond 函式

cond 函式提供程式多重分支的功能。其語法如下例：

```
cond（
    （condition1 exp11 exp12…）
    （condition2 exp21 exp22…）
    （condition3 exp31 exp32…）
    （t expN1 expN2…）
）；  cond
```

cond 函式會去計算每一個分支中的條件，直到它找到一個 *non-nil* 的分支。然後它會執行在此一分支下的所運算表式， 俟完成後離開此函式。 回傳值則是此一被執行的分支中的最後一個算式的結果。

cond 函式的作用相等於

```
   if        condition1    exp11    exp11    …
else if      condition1    exp21    exp22    …
else if      condition1    exp31    exp32    …
…
else         expN1 expN2….
```

以下是一個使用 cond 函式的例子：

```
Procedure（trClassify（signal））
  cond（
        （!signal nil）
        （!numberp（signal）nil）
        （signal >= 0 && signal < 3 ‘week’）
        （signal >= 3 && signal < 10 ‘midweate）
        （signal >= 10 ‘extreme）
        （t          ‘unexpected）
       ）  ;  cond
）; procedure
```

1.2 Iteration 函式

有兩個基本的 iteration 函式可用在 SKILL 語言中；*while* 和 *for*。它們兩者是使用率非常普遍的函式。

While 函式

While 函式是非常普遍的函式，因爲做每一件事都能使用到 while。

當你使用 while 函式時記得，所有的判斷條件在每一次迴圈時，都會被測試一次，所以如果有些控制條件的並不因迴圈的內容改變而有所改變， 就不應該將它們放在 while 迴圈的條件式之中，應該移出迴圈外。考慮下列的程式碼：

```
while（i < length（myList）
        …
        i++
）
```

在這迴圈的範圍內無法改變 *myList* 的值，使得每次迴圈都重新計算 *myList* 的動作毫無意義。所以最好是將 *length(myList)*的值在迴圈外先計算好指定一個變數，再進入迴圈執行並以變數來取代原來在判斷式中的 *length(myList)*。

當使用迴圈時，可以先考慮一下是否使用 foreach，setof，或 exists 等函式會更恰當。

For 函式

For 函式的最主要的優點就是它會自動宣告迴圈變數 ，換言之就是不須去用到宣告區域變數的結構，像 let 或 prog 函式。 另一方面也意謂著該變數無法在迴圈外使用，這就是與 C 不同之處。考慮下列的程式片斷：

```
for（ i   I   length (myList)
      EvaluateList（i）
）
if（i = = 0
      printf（” The list was empty!\n”）
）
```

if 函式在測試條件式時會產生錯誤，因爲此時變數 *i* 已經帶任何值了。

2. 選擇函數

在 SKILL 中有兩個選擇函式：*caseq* 和 *case*。這兩個函式之間的不同處在准許用測試條件範圍內的數值範圍。 *caseq* 爲 *case* 的較快版本， *caseq* 使用函式 *eq* 後有點類似，*caseq* 限制只接受符號和較小的整數常數（-256 <= *i* <= 255），或是只包含上述二者的串列。

caseq 和 *case* 函式允許串列出現在待測部份，假若比對測試值出現（*eq* 或 equal 串列中的元素）在串列之中，就是比對成功。

一個使用 caseq 函式常見的錯誤是：誤以爲在 caseq 函式的條件部份的值會被計算出來。考慮以下的呼叫 caseq：

```
caseq（x
      （’a “a”）
      （’b “b”）
）
```

這些的測試情況的部份， ‘a 和 ‘b 是不會被求值的，所以這些程式碼相等於

```
caseq（x
      （（quote a） “a”）
      （（quote b） “b”）
```

```
)
```

換言之，如果 x 的值是符號 a 或是符號 *quote*，caseq 會傳回值 “a”，很明顯這並不是原本要的結果。

當我們在決定測試情況的代表符號時必須要注意，不要使用 t，因爲它代表的是內定的情況，舉例來說，考慮此工作函式會傳回一個值 *t*、*nil* 或一個不確定的值。

它也許會寫一個如下列的函式

```
caseq（value
    （t              printf（”suceeded . \n”））
    （nil            printf（”failed. \n”））
    （indeterminate   printf（”indeterminate. \n”））
）
```

但是此函式將不會正確動作，因爲 *t* 情況是內定情況，所以一定會比對成功。此正確方法寫在如下：

```
caseq（value
      （nil              printf（”failed . \n”））
    （indeterminate   printf（”indeterminate. \n”））
    （t              when（eq（value， t）
                          printf（”succeded. \n”）
                     ）
            ）
）/* caseq */
```

避免這問題的另一方法是將 *t* 放在括弧之內，但此做法並不好，SKILL Lint 程式警告我們不當使用 *t* 情況。

3.以 prog 宣告區域變數

出現在 SKILL 程式中的所有變數內定都是全域變數，除非我們有特宣告其

爲區域變數。 我們可以使用 *prog* 控制結構來宣告區域變數，在進入 *prog* 執行時所有區域變數的值會設爲 *nil*，俟離開 *prog* 函式後再回復原來的值。

符號的目前值是無論何處都可以取得的， SKILL interpreter 使用堆疊的式來管理符號的值：

■ 符號目前的值是在堆疊的最上層。

■ 給符號一個值只會改變堆疊的最上層的值。

當你的程式呼叫 *prog* 函式時，此系統會將一個臨時值塞到各個區域變數的值堆疊上。當控制 *flow* 離開 *prog* 函式時，系統會將此臨時值取出，並恢復先前的值。

Prog 函式

prog 函式允許你使用 *go* 指令來在裡面撰寫迴圈的功能。除此之外， *prog* 也允許你使用多個 *return* 函式，來建立多重回傳點的目的。假若你沒有使用這兩個中的任一個特色，則建議使用 *let* 函式會更簡易並且更快速。

假若你須要條件式出口的 SKILL 的說明，可以使用 *prog* 函式。一個區域變數列和你的 SKILL 敘述，組成 *prog* 函式的引數。

```
Prog（ （local variables）your SKILL statements）
```

Return 函式

使用 *return* 函式去影響 *prog* 直接地回傳值並忽略後續的命令敘述，假若你沒有呼叫 *return* 函式，此 *prog* 式子會回傳 *nil* 值。

舉例說明： *trClassigy* 函式回傳值是 *nil*、*week*、*moderate*、*extreme*、或 *unexpected* ，取決於其上的信號。它並無使用任何的區域變數。

```
Procedure（trClassify（signal）
  Prog（ （）
      Unless（signal return（nil））
      Unless（numberpo（signal）return（nil））
      When（signal >= 0 && signal < 3return（’weak））
      When（signal >= && signal < 10 return（’moderate））
        When（signal>= 10return（’extreme））
        Return（’unexpected）
  ） ; prog
```

```
）; procedure
```

使用 *prog* 函式並使用 return 函式來提早從 for 迴圈離開，這個例子的目的是找出第一個小於或等於 10 奇整數。

```
Prog（ （）
  For（I 0 10
        When（oddp（I）
            Return（I）
                ）;  when
        ）;  for
  ）;  prog
```

4. Grouping 函式

三個主要函式 *prog*、*let*，和 *progn* 提供群組敘述的功能。除此之外， *let* 和 *prog* 函式允許宣告區域變數。此 *prog* 函式也是被允許使用 return 和 go 函式，go 函式與標記（label）一起應用可以在函式內部做到迴圈的功能。

以下是考慮利用 *prog*，*let*，和 *progn* 函式的那一個的時機：

- 假如不會用到區域變數和跳躍，使用 *progn*。
- 假如會用到區域變數但是不跳躍，則使用 *let*。
- 假如跳躍是需要的則使用 *prog*。

使用 prog、return、let

prog 命令應該只用在當它是絕對的需要時候，過度使用 *prog* 函式是造成 SKILL 程式碼效率低落的一大原因。從函式的程式碼中間跳回不但是付出很高的代價，也使得程式碼變得不易閱讀和了解。 因此大部份高階程式言都十鼓勵使用 go 函式（SKILL 的”goto”敘述）。

一個程式設計者通常使用 *prog* 型式，當有些錯誤檢查的動作必須在開始執行函式時完成，而函式剩下的程式碼只有在檢查動作通過後才會執行。 考慮下列的片斷程式碼：

```
procedure（check（arg1 agr2）
  prog（ （）
        when（illegal_val（arg1）
```

```
            printf（”Arg1 in error . \n”）
            return（ ）
        ）/ * end when * /
        when（illegal_val（arg2）
        printf（” Arg2 in error . \n”）
        return（ ）
        ）/ * end when * /
        Rest of code…
    ）/ * end prog * /
    ）/ * end check * /
```

去掉程式中的這些碼是合理的，除非有人很容易漏掉 *return* 的敘但要使用 *prog* 的函式。考慮下列另一種寫法：

這個 *procedure* 似乎較長，但是它是一個較清楚、快速、與易維護的碼：

```
procedure（check（arg1 arg2）
  cond（
      （illegal_val（arg1）
          printf（” Arg1 in error .\n”）
              nil
          ）/ * check arg1 * /
              （illegal_val（arg2）
                  printf（”Arg2 in error . \n”）
                  nil
          ）/ * check arg2 * /
        （t
          Rest of code…
        ）
  ）/* end cond * /
  ）/* end check * /
```

一個分離的小函式可在 *t* 條件的範圍中被呼叫，可以預期其引數是正確的。如此一來，只要花費小小的代價（額外呼叫一個函式），便可以將錯誤檢查的動作從函式本體中移走，使程式設計師可以更專注於函式的主要功能，而不用擔心其他周邊的事情。

另一個關於使用 *prog* 和 *let* 函式常犯的錯誤是區域變數的值初始化成 *nil*。所有 *prog* 和 *let* 函式的區域變數會自動初始化成 *nil*，記得 *let* 函式允許區域變數在宣告時設定初值，這可以節省時間和空間。而且只要注意程式撰寫的佈局，程式的可讀性不會減少多少的。

```
Procedure（testr（）
        Let（  ((localvar1 initvalue1）
                （localvar2 initvalue2）
                  localvar3                    /* initial value nil * /
        ）
        rest of code…
        ）/* end let */
）/* end procedure * /
```

當在宣告範圍內設定初始值時，無法參考其他的個區域變數。例如，下面的例子是錯誤的：

```
procedure（incorrect（list）
  let（  ((listhead        car（list））
           （headval       car（listhead））  / * wrong !!! * /
           ）
          rest of code…
```

注意此 *prog* 和 *let* 函式有不同的傳回值。

- 此 *prog* 函式的傳回值是指定在 *return* 敘述中，假如沒有 *return* 時則回傳 *nil*。
- 此 *let* 函式總是回傳最後一個命令敘述的回傳值。

使用 progn 函式

progn 函式是提供一種簡易的方法來群組命令敘述，在此群組中很多敘述都被要求要的，但只有一個會被執行。 *setof* 函式就是一例，它只允許單一的敘在條件部分。

使用 prog1 和 prog2 函式

Prog1 函式

Prog1 回傳第一個敘述的值。

```
Prog1
        X = 5
        Y = 7 )
⇨  5
```

prog2 函式

prog2 回傳第二個敘述的值。

```
Prog2 (
        X = 4
        P = 12
        X = 6）
= > 12
```

下面的例子說明 *prog2* 的優點：

```
Procedure（main（）
    Let（ （status）
        Initialize（）
        / * main body of program * /
        status = analyzedata（）
        wrapup（）
/ * return the status    */
    status
  ）/ * end let * /
）  /* end main * /
```

使用 *prog2* 這些程式碼可被更有效的執行：

```
Procedure( main（）
Prog2(
        Initialize（）
        / * main body of program.
        * the value of this will be returned by prog2
```

```
                * /

                analyzedata（）
                wrapup（）
    ）/* end prog2 * /
）/* end main * /
```

7

I/O 與檔案處理

1. 檔案系統介面

所有在 SKILL 的輸入和輸出都是根據 UNIX 的檔案系統來定義的，在 SKILL 中要寫一個 I/O 的命令敘述需要瞭解檔案、目錄和路徑。

檔案

一個檔案包含了資料，通常由多筆記錄所構成， 檔案會記錄幾個性質，包括名稱、建檔日期、上次存取時間等。 device 是一種特殊的檔案，有特別的性。

目錄

目錄都有一個名字， 就像是檔案一樣，但它包含了一串其他檔案， 目錄能被建立成許多階層，目錄使得相關的檔案能群集在一起。 因爲成千上萬個檔案能被存放在一個磁碟中，如果沒有目錄來整理存放，豈不大亂！ 還有網路上的檔案系統也必須依賴目錄的機制。

1.1 目錄路徑

通常你希望能依序去搜尋幾個特定的目錄，於是記錄其目錄路徑。 你經由相對的或是絕對地指定路徑來取得一個檔案。

在以下的描述所使用的“路徑”就是一般的名詞，檔案或目錄皆可適用。事實上在 UNIX 之下目錄不過是特殊形式的檔案。檔案名稱通常被視爲路徑的同義詞。

絕對路徑

你能夠用倒斜線（/）來指定路徑。當倒斜線出現在路徑名稱的第一個字時，代表系統的根目錄，中間層的目錄以倒斜線來做分隔字元。

相對路徑

任何路徑不是以“/”開始者，都是相對路徑。假如路徑開始是倒斜線（~/），則從你的根目錄中開始開始尋路徑 。假如下列的名稱中 ， 所解釋的使用者名稱是 ~jones/file1 說明檔案名字在 jones 的根目錄中。

假如路徑所開始的時期和倒斜線（ ./ ），在搜尋開始的目前的工作目錄。假如這些檔案名稱開始有兩個句點和倒斜線（ ../），開始搜尋目前工作目錄的上一層目錄。

SKILL 路徑

SKILL 提供一個彈性的機制來使用相對路徑，SKILL 定義一個路徑的串列稱爲 SKILL path，很多跟檔案有關的函式會用到此一串列。

路徑第一個字的重要性

當一個相對路徑不是以 ~或 ./開頭，吾人將此路徑傳入一個函式，則 SKILL 路徑會被用來當做目錄名稱， 加到該路徑之前以構成可能之絕對路徑。 函式 setSKILLPath 和 getSKILLPath 可以存取並且改變這個內部的 SKILL 路徑設定。

路徑順序的重要性

當相同檔案的名稱在多個路徑中出現，此時在 SKILL 路徑中的路徑的順序就非常重要的。所以要按照檔案存放的位置來考慮路徑設定的順序。

當一個檔案第一次被修改時 ，搜尋路徑的順序的設定也重要 ，如果你開啓一個輸出檔的同時， SKILL 會檢查所有 SKILL 路徑串列內的路徑，找尋是否有同名的檔案，若有，則會開啓第一個找到的同名檔案。如果找不到 ，系統會

建立一個新檔案於第一順位的目錄裡。

了解你的 SKILL 路徑

有了 SKILL 路徑的內部串列，提供一個有力省時工具。但相對地也有可能造成混亂來源，當有所懷疑時 ，重複檢查 SKILL 路徑的設定或是乾脆把它設成 *nil*。

當你起動你的系統時，SKILL 路徑會設定成內定值，你可以使用 *setSKILLPath* 函式去確認路徑有設定正確。

1.2 應用 SKILL Path 工作

設定 SKILL Path（setSKILLPath）

函式 *SetSKILLPath* 可用來設定內部的 SKILL 路徑，你可以一組空白字隔開的字串，或是一個字串的串列來指定目錄路徑。 假如所有的目錄路徑存在，則此函式傳回 nil 值 。假如有任何路徑不存在，它將回傳一個串列記錄每一個無效的路徑。

SetSKILLPath（’（” .” “~” “~/cpu/test1”））

⇨ nil ; if “~cpu/test1” exists.

⇨ （” ~/cpu/test1 “） ; if “~/cpu/test1 “does not exist.

下面的函式呼叫作用相同：

SetSKILLPath（” . ~ ~ /cpu/test1”）

傳回目前的 SKILL path 設定（getSKILLPath）

函式 getSKILLPath 傳回目前的 SKILL path 所記載的所有路徑，結果如下例所示：

setSkillPath（’（”.” “~” “~cpu/test1”））　⇨ nil

getSkillPath（） ⇨（”.” “~” “~/cpu/test1”）

以下舉例說明，顯示如何去加一個目錄在你搜尋路徑開始（定義目錄爲 “ ~/lib “）

setSkillPath（cons（” ~/lib “ getSkillPath（））） ⇨ nil

getSkillPath（） ⇨ （”~/lib “ “ . ” “~” “~/cpu/test1”）

1.3 系統安裝路徑

找出安裝路徑（getInstallPath）

函式 *getInstallPath* 會回傳系統安裝路徑（那是指 Cadence 產品安裝時指定的根目錄），回傳的值是單一字串構成的串列，而且其中的每一個路徑都是以絕對路徑的形式呈現。

getInstallPath（） ⇨ （”/usr/cds/4.2”）

重新設定安裝路徑（prependInstallPath）

函式 *prependInstallPath* 會“假裝”Cadence 安裝路徑是給定路徑（假如需要則可能加一個倒斜線在前面），並將其以字串形式回傳。舉例如下：

getInstallPath（） ⇨ （” /usr5/cds/4.2”）

假設這是你的安裝路徑

prependInstallPath（”etc/context”） ⇨ ”/usr5/cds/4.2/etc/context”

一個 slash 被加入

prependInstallPath（” /bin”） ⇨ ”/usr5/cds/4.2/bin”

```
setSkillPath（list（”.” PrependInstallPath（”bin”）
                  prependInstallPath（”etc/context”）））
    ⇨ nil
```

找出一個目錄階層的根目錄（cdsGetInstaPath）

函式 *getInstallPath* 傳回此 *dfll* 階層的根，然而函式 *cdsGetInstPath* 傳回整個階層的根。 *cdsGetInsPath* 較爲普遍化，主要是給所有的 *dfll* 和 *non-fdll* 應用。舉例說明：

getInstallPath（） ⇨（”/usr/mnt/hamilton/9304/tools/dfII”）

cdsGetInstPath（） ⇨“ /usr/mnt/hamilton/9304”

1.4 檢查檔案狀況

以下的函式的引數若是採用相對路徑的表示方式，則參考目前的 SKILL path 以決定其絕對路徑，不過 SKILL path 必須是 *non-nil* 的。

檢查檔案是否存在（isFile ，isFileName）

isFileName 函式用以檢查檔案是否存在，此檔案名稱可以絕對或相對路徑來給定。假如發現此名稱能屬於任一個檔案或是路徑，則 *isFileName* 傳回值 *t*。*isFile* 的功能類似，只是對目錄的處理不同。舉例如下：

isFileName（" myLib"） ⇨ t

一個目錄是一個特殊檔案型式

isFileName（"triade"） ⇨ t

isFileName（" triadl"） ⇨ nil ;triad1 非現行的工作目錄結果

假如檔案存在則 *isFile* 檢查結果與 *isFileName* 完全相同，假存在的是一個目錄，則 *isFile* 的回傳值變成 *nil*。

檢查一路徑是否是爲目錄（isDir）

假如要檢查一路徑是否存在且是否爲目錄名稱，則呼叫 *isDir* 函式。當給定的路徑引數是相對路徑時，則參考目前的 SKILL path 以決定其絕對路徑，不過 SKILL path 必須是 *non-nil* 的。

isDir（"myLib"） ⇨ t

isDir（" triadc"） ⇨ nil

其中 *myLib* 是目錄而 *triadc* 是檔案。

檢查有無權限讀檔案（isReadable）

isReadable 會檢查你是否有權讀某一檔案或是目錄的內容。舉例如下：

isReadable（". /"） ⇨ t

假如目前的工作目錄是可讀的。

isReadable（"~/mylib"） ⇨ nil

假如 ~/myLib 並不可讀或不存在。

檢查有無權限修改檔案（isWritable）

函式 isWritable 會檢查你是否有權去修改某一檔案或是目錄的內容。舉例如下：

isWritable（"/tmp"） ⇨ t

isWritable（”~/test/out.1”） ⇨ nil

假如 *out.1* 並不存在或沒有寫入它的權力。

檢查有無權限執行檔案（isExcutable）

函式 *isExecutable* 會檢查你有無執行檔案或搜尋目錄的權力。假如它允許你目錄的名稱在你的 UNIX 路徑搜尋檔案中是可執行的，則稱此目錄是可執行的。舉例如下：

isExecutable（”/bin/ls”） ⇨ t

isExecutable（”/usr/tmp”） ⇨ t

isExecutable（”attachFiles”） ⇨ nil

結果顯示 *attachFiles* 並不存在或不可執行。

決定檔案大小（fileLength）

函式 *fileLength* 決定在檔案中的位元數，一個目錄在此事件中看似就像一個檔案。舉例如下：

fileLength（” /tmp”） ⇨ 1024

此回傳的結果是依系統現況而定。

fileLength（”~/test/out.1”） ⇨ 32157

此例子假設檔案存在，假如此檔案並不存在，你會得到錯誤的訊息像是

*Error * fileLength : no such file or directory - “ ~/test/out.1 ”

取得已開啓檔案的資訊（numOpenFiles）

函式 *numOpenFiles* 會傳回此已開啓檔案的數目，以及一個行程（process）能開啓的檔案最大數目，回傳值是一個包含兩個元素的串列。

numOpenFiles（） ⇨ （6 64）

回傳的結果是依系統現況而定。

f = infile（”/dev/null”） ⇨ port : “/dev/null”

numOpenFiles（） ⇨ （7 64）

又多一個檔案被開啓了。

應用檔案指標與檔首之偏移值（fileTell ，fileSeek）

對於一個已開埠(port)的檔案，函式 *fileTell* 會傳回目前檔案指標距離檔首之位元數。函式 *fileSeek* 則是設定下一個運算要從檔案中的那一個位置開始處理。此位置以位元數來表示， *fileSeek* 有三個引數，第一個引數是使用之埠(port)，第二個引數是以位元數表示是向前移動之偏移量（負數代表是向後移），第三個引數的有效值則是：

0 由檔首開始計算偏移值
1 自目前檔案指標所在位置開始計算偏移值
2 由檔尾開始計算偏移值

假設檔案 test.dat 只包含一行文字如下：

0123456789 test.xyz

以下是程式範例：

```
p = infile（”test.data”）        ⇨  port   : “test.data”
fileTell（p）                    ⇨   0
for（I 1 10 getc(p)）            ⇨   t     ; skip first 10 characters
fileTell（p）                    ⇨  10

fscanf（p   “%s” s）             ⇨   1     ; s = ”test” now
fileTell (p)                     ⇨  15

fileSeek ( p 0 0)                ⇨  t
fscanf( p “%d” x)                ⇨  l      ; x = 123456789 now

fileSeek（p 6 1）                ⇨  t
fscanf（p “%s” s）               ⇨  l      ; s = “xyz” now

fileSeek（p –12 2）              ⇨  t
fscanf ( p “%d” x)               ⇨  1      ; x = 89 now
```

1.5 使用目錄

以下的函式的引數若是採用相對路徑的表示方式，則參考目前的 SKILL path 以決定其絕對路徑。

建立目錄（createDir）

函式 *createDir* 可以建立一個新的目錄，所給的目錄名稱引數可以是用絕對或相對路徑表示。假如無法執行此一函式，表示你並無權去異動其父目錄或此父目錄根本就不存在。

createDir（”/usr/tmp/test ”） ⇨ t
createDir（”/usr/tmp/test”） ⇨ nil

第二次呼叫不成功因爲此目錄已經存在。

刪除目錄（deleteDir）

函式 *deleteDir* 可以用來刪除一個目錄，此自錄名稱能以絕對或相對路徑來給定。 假如你沒有權限去刪除檔案或是嘗試刪除非空的目錄，則你會得到錯誤的訊息。

createDir（”/use/tmp/test”） ⇨ t
deleteDir（”/use/tmp/test”） ⇨ t
deleteDir（”/usr/bin”）

假如你並無允許去刪除 /bin，你會得到一個關於違反使用權限的錯誤訊息。

刪除檔案（deleteFile）

函式 *deleteFile* 可用來刪除檔案，此檔案名稱可以絕對或相對路徑表示。假如傳入的引數是一個檔案鏈結（link），則被刪除的是此一鏈結檔本身， 而非所鏈結的對象。

deleteFile（”~/test/out.1”） ⇨ t

假如這些檔案存在且可被刪除的，

deleteFile（”~/test/out.2”） ⇨ nil

假如這些檔案並不存在，

deleteFile（”/bin/ls”）

假設你沒有使用 /bin 的權限，你會得到一個關於違反使用權限的錯誤訊息。

建立一個唯一的檔案名稱（makeTempFileName）

函式 *makeTempFileName* 會附加一個字尾字串給一個路徑樣板以產生一個獨一無二的檔案名稱。連續的呼叫 *makeTempFileName* 會回傳不同的結果，實

際上假如第一個名稱回傳使用創造一個檔案，在第二次呼叫前在相同的目錄中。

■ 合成路徑的最後一個部份（即檔名）保證不多於 14 字，以下舉例要求的 15 個字元的例子。

makeTempFileName（”/tmp/123456789123456”）

⇨ “/tmp/12345678a08717”

■ 假如原始的參考檔名很長，它會從最後一字往前來縮減。

MakeTempFileName（”/tmp/123456789.123456”）

⇨ “/tmp/12345678a08717”

■ 在新的附加字尾之前任何的追蹤 Xs 是從 template 來移除。

你應該遵守傳統習慣將臨時檔案放置於 /tmp 目錄中。

d = **makeTempFileName**（”/tmp/testXXXX”） ⇨ “/tmp/testa00324”

標出 Xs 已被移除。

createDir(d) ⇨ t

名稱是被使用在此時的。

makeTempFileName（”/tmp/test”） ⇨ “/tmp/testb00324”

此時一個新的名稱被傳回。

列出所有檔案和目錄名稱（getDirFiles）

函式 *getDirFiles* 會列出在一個目錄之下的所有檔案和子目錄（包括 . 和 ..）。舉例如下：

getDirFiles（car（getInatallPath（））） ⇨

（”.” “..” “bin” “cdsuser” “etc” “group” “include” “lib” “pvt”

“samples” “share” “test” “tools” “man” “local” ）

將檔案名稱展開成絕對路徑（simplify Filename）

函式 simplifyFilename 回傳一個檔案的完全展開路徑名稱。多餘的倒斜線會被移除，程式鍵結也會被實際位置取代。如下例：

simplifyFilename（”~/test”） ⇨ “/usr/mnt/user/test”

假設此使用者的根目錄是 /usr/mnt/user。

取得目前的工作目錄（getWorkingDir）

函式 *getWorkingDir* 回傳目前的工作目錄。如果此結果顯示成“*~prefixed*”型式。舉例說明，假設 user1 的工作目錄是 /usr/mnt/user1，而目前的目錄是 /usr/mnt/user1/test1，如果可以的話，回傳值會優先以 ~/test 形式顯示，不行的話才會用 /usr/mnt/user1/test。

getWorkingDir（）⇨ “~/project/cpu/layout”

改變目前的工作目錄（changeWorkingDir）

函式 changeWorkingDir 能改變工作目錄成你指定的目錄。此名稱可以相對或絕對路徑表示。如果你提供的是一個相對路徑，則此路徑會參考 UNIX 作業系統裡面的 cdpath 變數,而不是 SKILL 的路徑。

警告：

使用此函式要小心：假如 “.”是 SKILL 路徑或是 libraryPath 的一部分，則改變工作目錄會影響 SKILL 檔案或設計資料的能見度。

假設有一個目錄 /usr5/design/cpu，並且有開放適當的使用權限，假設在 /usr5/design/cpu 之下沒有 test 這個目錄。

changeWorkingDir（”/usr5/design/cpu”） ⇨ t
changeWorkingDir（”test”）

第二個函式執行結果，出現路徑不存在的錯誤訊號。

2. 埠

在 SKILL 中所有輸入與輸出的資料都要透過一個特殊的資料型態叫做埠（port），一個埠能被開啓成可讀（輸入埠）或是可寫（輸出埠），在 SKILL 中的埠是類似於 C 語言的 FILE 變數。

大部分的 UNIX 系統都會限制在任何時間中，能同時開啓的檔案數目（典型標準在 30 到 64 之間）。

你的的應用程式有時會用到這些罕用的檔案描述子， 而且只有少數的埠可

以使用，所以你應總是關掉埠已不用的 I/O 埠，以避免用光系統提供的輸出入埠。如此可免當你的程式移植到另一台 UNIX 的機器中時，無法執行。

2.1 Predefined Ports

在 SKILL 中大部分的 I/O 函式都可使用變數當做引數，假如未指定輸出入埠，則內定使用 *piport* 和 *poport* 做爲輸入埠和輸出埠。在下列表格中列出 SKILL 內建的輸出輸入埠。

警告：

假如有必要，你可以重新定義內定值，但很多 SKILL 內建函式是使用特殊的埠，而且這種連線是與硬體有關的，所以你應該小心不要定義任何不合法的值。

此外 *stdin*、*stdout*、*stderr* 埠也是內建好的輸出入埠，這些埠相當於標準輸入、標準輸出、標準錯誤，可用在每一個 UNIX 程式中。

預建好的輸入輸出埠

Name	usage
piport	標準的輸入埠，類似於並初始化成 C 語言中的 stdin。
poport	標準的輸出埠，類似於並初始化成 C 語言中的 stdout。
errport	列印錯誤訊息的輸出埠，類似於並初始化成 C 語言中的 stderr。
ptport	對於列印追蹤資訊的輸出埠，初始化成 C 語言中的 stderr。

2.2 開啓與關閉輸出入埠

以下的函式用於開啓或關閉輸入輸出埠。 二者都是參考 SKILL 路徑變數。

開啓一個輸入埠以讀檔（infile）

函式 infile 開啓一個輸入埠以供你讀取指定的檔案內容。此檔案名稱能以

絕對或相對路徑的方式來給定。

infile（”~/test/input.il”） ⇨ port : “~/test/input.il”

假設這個的檔案存在並是可讀的。

infile（”myfile”）= > nil

假設 *myfile* 根據 SKILL 路徑它不存在或是不可讀。

開啓一個輸出埠以存檔（outfile）

函式 *outfile* 是一個由你所指定儲存輸出的檔案,這個檔案的名稱可由絕對路徑或相對路徑來指定。

- 如果給的是相對路徑，而且 SKILL 路徑的值不是 nil，則 SKILL 路徑內的所有目錄路徑就會被依序被檢查。
- 如有找一個同名的檔案，則系統會取第一個出現的同名檔案做爲覆寫的對象。
- 如果沒有可更新的檔案， 它就在第一個允許寫入的目錄內， 建立一個以該名稱的新檔。

p = outfile（”~/test/out.il” “w”） ⇨ port:”~/test/out.il”

傳回輸出埠的名稱去寫入那個檔案。

outfile（”/bin/ls”） ⇨ nil

如果檔案爲了不能寫入而無法開啓，則傳回 *nil*。

將輸出緩衝區所有的字元寫出去（drain）

函式 *drain* 會將輸出緩衝區內所有的字元寫出去輸出埠。 *drain* 類似於在 C 語言中 *fflush* 與 *fsync* 的組合。如果埠到 drain 是一個輸入方向或是已經被關閉，則你會得到一個錯誤的訊息。

drain（） ⇨ t

drain（poport） ⇨ t

流出，關閉和釋放的埠（close）

當一個埠要被關閉時，它會釋出緩衝區資料，關閉埠，最後釋放掉這個埠。這同時也釋出了關連於這個埠的檔案指標。不要使用這個函式來處理 piport，poport，stdin，stdout 及 stderr 等。

p = outfile（”~/test/myFile”） ⇨ port:”~/test/myFile”

close（p） ⇨ t

3. 輸出

SKILL 提供一些函式可供未格式化和格式化的資料輸出。

3.1 未格式化的輸出

以內定格式印出運算表式的值（**print, println**）

函式 *print* 會以內定格式印出運算表式的值（例如，strings 會用雙引號括起來）。

print（”hello”）
“hello”
⇨ nil

顯示到 *poport* 並傳回 *nil*。

函式 *println* 列印的方式與函式 *print* 相同，只是在輸出的結果之後再多印一個換行字元。 *println* 在印出每一個換行字元後清除輸出埠的緩衝區資料。

println（”Hello World!”）
“Hello World!”
⇨ nil

印出一個新行（\n）字元（newline）

列印一個換行控制字元，如果你沒有指定輸出埠，它就輸出到 *poport* 這個標準的輸出埠。在印完新進的字元後，newline 函式會清除輸出埠的緩衝區資料。

print（”Hello”）**newline**（ ）print（”World!”）
“Hello”
“World!”
⇨ nil

列印一個有限制元素個數和層次的巢狀目錄（printlev）

函式 *printlev* 可以列印一個有限制數目和層次的巢的目錄。如果你沒有指定個數和階層數，目錄會完全的被列印，不論有多少的元素或有幾層的目錄。

應用程式透過 *printlev* 函式列印，可以控制列印串列時的元素個數限制，或最多可印幾層。這樣便可以限制交談式輸出的資料量。也可以避免去列印到無限循環串列（初學者由於不當使用破壞性目錄修改函式，很容易產生的），而進入無限迴圈。 *printlev* 使用下列語法：

printlev（g_value x_level x_length [p_outputPort]） ⇨ nil

兩個整數變數分別代表印列長度及層次（指定 *x_length* 和 *x_level*），控制所要列印巢內元素和層次的數目。目錄元素超過最大列印長度時可縮寫成 "…"，目錄的層比最大多層次還多時也可縮寫成 "**&.**"，在 SKILL 內，列印長度和列印層次之初始值爲 *nil*（意指沒有強迫限制），但是每一種應用可以設定它自己的限制。

這個 *printlev* 函式和 print 函式的語法除了它加入指定最大層次和長度這兩個參數外，其餘都是一樣。

```
List = '(1 2 (3 (4 (5))) 6)
printlev(List 100 2)
(1 2 …)
⇨ nil
printlev(List 3 100)
(1 2 (3 (4 &)) 6)
⇨ nil
```

假設埠 p 存在，印到埠 p。

3.2 格式化輸出

你可以在列印格式化字元之前指定其欄位範圍的寬度。例如， %5d 會印出一個寬爲５格的整數。 如果寬度的起始爲值爲 " 0 " ，則後續的空格會填上０而不是空白。對於格式字元 f 及 e ， 指定的列印寬度值可以再附上 "．" 和一個整數指定其精確度，那是在小數點之可列印的位數。

輸出欄位的內容內定是向右靠齊，除非在字元 "%"後立即接一個負符號 "-"，才會向左靠齊。要印出一個百分比符號，你必須連續使用兩個百分比符號，你必須在你的格式字串中加入 "\n"，列印的資料才會跳行，至於加入 "\t" 控制字元相當於定位鍵的作用。

共同輸出格式規格

格式規格	引數型式	列印
%d	定點數	以十爲基底的整數
%o	定點數	以八爲基底的整數
%x	定點數	以 16 爲基底的整數
%f	浮點數	浮點數 [-]ddd.ddd
%e	浮點數	浮點數 [-]d.ddde[-]ddd
%g	浮點數	選用參數 f 或 e 印出，視何者在使用完全的精確度情況下，佔用欄寬較小者
%s	字串符號	印一個沒有引號的字串，或印出一個符號的名字
%c	字串符號	第一個字元
%n	整數，浮點數	數目
%L	目錄	資料型態的內定格式
%P	目錄	點
%B	目錄	方框（box）

對於格式化的輸出，SKILL 借重 C 語言中常用的 I/O 函式 printf，fprintf 及 sprintf 的強大功能。SKILL 對於這些使用這些函式提供一個有用的介面。茲介紹如下：

格式化的輸出寫到 poport（printf）

函式 *printf* 將格式化的輸出寫到 *poport*，可選擇的參數項目根據其對應的格式字串的描述，被列印出來。 *printf* 除了沒有使用埠的參數和輸出被寫到 *poport* 外，其餘功能與 *fprintf* 相同。

```
x = 197.9687
printf（”The test measures %10.2f.\n” x）
```

列印下面那一行到 *poport* 並傳回 t。

The test measures 197.97. ⇨ t

輸出格式化資料到一個埠（fprintf）

函式 *fprintf* 以第一個參數指定一個埠，其格式化列印的方式與 *printf* 類似。

```
x = 197.9687
fprintf（p “The test measures %10.2f.\n” x）
```

列印下面那一行到埠 *p* 且傳回 t。

The test measures 197.97. ⇨ t

寫格式化輸出到一個字串變數（sprintf）

函式 *sprintf* 是將格式輸出的字串存在第一個引數變數內。如果第一個參數是空的那就不做任何事情，格式化字串被傳回。在 *sprintf* 因爲有內部緩衝區，*sprintf* 可控制多少字元是有一個限制的，但是這個限制是非常的大（8192 字元）使得它通常不會有問題。

```
sprintf（st “Memorize %s number %d!” “transaction” 5）
⇨ “Memorize transaction number 5!”
st
    ⇨ “Memorize transaction number 5!”
p = outfile（sprintf(nil “test%d.out” 10)）
    ⇨ port:”test10.out”
```

美化列印

在文件表格內， SKILL 對於“美化列印”提供一些函式，設定適合的精確度和長的資料目錄去使得它們更易讀和更容易操作。

你需要 SKILL 發展環境許可證去美化列印函式。

你可以使用下面的函式精確度（pp）。

美化列印函式定義（pp）

函式 *pp* 可以美化函式定義的列印，這個函式必須是一個可讀的函式（二進位碼的函式不能被美化列印），每一個函式的定義被列印出，如此它可以再被讀回進入 SKILL 。 *pp* 不會計算它的第一次參數，但它會計算第二次參數，如果有給定的話。

```
procedure（fac(n) if(n <= 1 1 n*fac(n?)）  ⇨ fac
    pp fac
    procedure(fac(n)
        if(n <= 1) 1
        （n * fac(n ?1)））
```

```
            )
        )
        ⇨  nil
```

定義階層函式，然後美化列印它到 *poport*。

美化列印長的資料串列（pprint）

函式 *pprint* 除了會試著盡可能地美化列印每一個輸出值之外，其餘的功能與 *print* 相同。（*pprint* 的工作與 *pp* 函式不一樣，*pp* 是一個 *nlambda* 函式，只讀取一個函式的名稱；然而 *pprint* 是一個 *lambda* 函式，可以讀取 SKILL 的任何物件。

這個 *pprint* 函式是很好用的，例如，當須要列印一個很長的串列，而 print 只是簡單列印一（可能很大）行，但是如果需要， *pprint* 會限制單一行列印的字數，並產生多行列印輸出。這個多行列印輸出讓未來要做輸入動作時更容易。

```
pprint  '(1 2 3 4 5 6 7 8 9 0 a b c d e f g h i j k)
(1 2 3 4 5
    6 7 8 9 0
    a b c d e
    f g h i j
    k
)
⇨  nil
```

4. 輸入

當描述輸入函式，這個手冊常常使用一個術語“form”來視作一個邏輯上的輸入單元。一個 form 可以是一個表示式，譬如一些原始碼或可展開到多重輸入行的資料串列。 輸入函式每次只讀入一行，會持續地讀直到找到但如果在每行的末端找不到一個完整的“form 結束”（complete-form）符號就會繼續地讀。你可以想像輸入函式和 SKILL 函式在下列的方法如何動作。

輸入函式

輸入來源	SKILL 會計算	SKILL 不會計算	特定應用格式
File	load	lineard	infile
	loadi		gets
			getc
			fscant
			close
String	evalstring	linereadstring	instring
	Loadstring		gets
	Errsetstring		getc
			fscanf
			close

SKILLform 從檔案讀出的內容可以是計算過的或未計算過的。輸入字串可以有一個它們自己所擁有的特定用途語法。程式設計師的責任是去開啓一個埠，了解和應用特殊語法，處理輸入，然後關閉埠。

4.1 讀取和計算 SKILL 格式

讀取和計算一個字串的表達式（evalstring）

函式 evalstring 讀取和計算存放在字串裡的的表示式，結果會被傳回。舉例如下：

evalstring(“1 + 2”) ⇨ 3
evalstring(“cons(‘a ‘(b c))”) ⇨ (a b c)
car ‘(1 2 3) ⇨ 1
evalstring(“car ‘(1 2 3)”)

SKILL 會示警 *car* 是一個未帶值的變數。

開啓一個字串並執行它的表示式（loadstring）

函式 *loadstrin* 會開啓一個字串來讀取，然後分析語法和執行儲存在字串內的表示式，正如 *load* 函式對檔案讀取處理一樣。 *loadstring* 有兩點是不同於 evalstring 的。

- 使用 *lineread* 的模式
- 如果計算成功，則傳回 t 值

loadstring "1+2" ⇨ t
loadstring "procedure(f(n) x=x+n)" ⇨ t
loadstring "x=10\n f 20\n f 30" ⇨ t
x ⇨ 60

讀取和計算一個表示式並檢查錯誤（errsetstring）

函式 *errsetstring* 讀取並計算一個存在字串裡的表示式。*errsetstring* 除了會呼叫 *errset* 去捕捉在分析與計算期間可能發生的錯誤外，基本上與呼叫函式 *evalstring* 的作法雷同。

errsetstring("1+2") ⇨ (3)
errsetstring("1+'a") ⇨ nil

如果發生一個錯誤則傳回零

errsetstring("1+'a" t) ⇨ nil

印出一個錯誤訊息

Error plus: can't handle (1 + a)

載入檔案（load, loadi）

函式 *load* 會開啓一個檔案，重覆呼叫 *lineread* 去讀進一行資料，並且立即計算每個讀進來的 form 格式。當檔案的末端到達時就關閉檔案。除非發現錯誤，否則檔案會安靜地被讀取。如果載入被 control-c 中斷，則函式會跳過正在載入的檔案。

SKILL 有自動載入的特性，它允許應用程式載入所要求的函式到 SKILL 裡。如果一個正在執行的函式尚未被定義，SKILL 會檢查此一函式的屬性中是否有一個 *autoload* 的屬性。如果有的話，這個值不是一個字串就是一個傳回字串的函式，此一字串即是要被載入的檔案名稱。這個檔案應該包括可以觸發 *autoload* 動作的函式定義。整個自動載入的動作是很明顯。

```
load("testfns.il")                    ; Load file testfns.il
fn.autoload = "myfunc.il"             ; Declares an autoload
property.
fn(1)
```

在這個點，函式 *fn* 是沒有被定義的，如此觸發一個包括 *fn* 的定義的 *myfunc.il* 的自動載入檔。

```
fn(2)          ; fn 是現在是已定義而且正規的執行
```

函式 *loadi* 大致與 *load* 動作相同，只是會忽略在載入的期間內所發生的錯誤，只印出錯誤訊息，然後續繼載入動作。

loadi(“testfns.il”)
Loads the testfns.il file.
loadi(“/tmp/test.il”)

從目錄 *tmp* 載入 *test.il*。

4.2 讀取但不計算 SKILL 格式

分析從輸入埠取得的一行資料（lineread）

函式 *lineread* 分析從輸入埠取得的一行資料，然後存入一個串列使你能更進一步處理之。通常是在直譯器的最上層動作會去呼叫 lineread，並且只能針對 SKILL 語法來分析。

只讀入一行輸入資料除非第一行的結尾少了必要的左括號， 或者二元運算子的右側運算元不見，才會再讀下一行，直到上述情況不再發生爲止。 如果讀到一空白行， *lineread* 會回傳 t 值； 如果讀到檔案的末端，會傳回 *nil*。

```
lineread(piport)            ；讀入下一行的表達式
f 1 2 +                     ；開始讀入檔案的第一行
3                           ；第二行輸入
⇨  f(1 (2 + 3))
lineread(piport)
f(a b c)                    ；檔案內其它行的輸入
⇨  ((f a b c))              ；傳回輸入目地的目錄
```

讀一字串到目錄內（linereadstring）

函式 *linereadstring* 執行 lineread 來處理最近讀入的字串，並且傳回讀入的 form。在第一個 form 之後，任何東西是被忽略。

```
linereadstring “abc”              ⇨  （abc）
linereadstring “f a b c”          ⇨  （f a b c）
linereadstring “x + y”            ⇨  （（x + y））
linereadstring “f a b c\n g 1 2 3”  ⇨  （f a b c）
```

在最後的例子，只有第一個 form 被讀入。

4.3 讀取特定用途的格式

當你必須讀入一個檔案的內容，而該檔案並非採用 SKILL 相容的格式時，在下列的輸入函式很幫助的。

讀入被格式化的輸入（fscanf）

函式 *fscanf* 根據其格式定義的內容，從一個埠去讀入格式化的字串，而結果被存在對應的變數。 *fscanf* 可以視爲 *fprintf* 輸出函式的相反。*fscanf* 傳回與和它的格式字串成功地配對的輸入項目。如果它遇到檔案的結尾則傳回 *nil*。

對於字串變數，其輸入字串最大的尺寸是 8k。在讀入 SKILL 物件時，函式 *lineread* 比 *fscanf* 要快些。這個輸入格式接受 *fscanf* 如下面的概述。

一般輸入格式

格式	引數的型態	尋找標的
%d	定點數	一個整數
%f	浮點數	浮點數
%s	字串	輸入一個字串（限定其範圍）

fscanf（p “**%d %f**” i d）

從輸入埠 *p* 讀入一個整數和一個浮點數，各別儲存到變數 *i* 及 *d* 。
假設在一個稱爲 *testcase* 的檔案內有一行：
hello 2 3 world

```
x = infile("testcase")                    ⇨  port:"testcase"
fscanf( x "%s %d %d %s" a b c d )         ⇨  4
(list a b c d)                            ⇨  ("hello" 2 3 "world")
```

讀入一行且儲存在一個變數（gets）

函式 gets 從輸入埠讀入一行且以字串型態儲存在變數內。這個字串也是 gets 的回傳值。在字串內，跳行字元將是其最後一個字元。如果 EOF 被相遇且變數保持最後的值，則 gets 傳回 *nil*。假設檔案 test1.data 有下列兩行資料：

```
#This is the data for test1
0001 1100 1011 0111
```

```
p = infile("test1.data")   ⇨   port:"test1.data"
gets(s p)                  ⇨   "#This is the data for test1\n"
gets(s p)                  ⇨   "0001 1100 1011 0111\n"
s                          ⇨   "0001 1100 1011 0111\n"
```

從一個輸入埠讀入並傳回單一字元（getc）

gets 從輸入埠讀入單一字元並回傳之，如果字元是一個不可列印的字元，它的八進制的值會被存成符號。如果你是使用 C 語言，那你應該注意到 *getc* 和 SKILL 的函式 *getchar* 是完全地不相關，如果 EOF 被相遇，則 *getc* 傳回 *nil*。對 *gets* 和 *getc* 而言，輸入埠的引數是可有可無的，如果未指定輸入埠，則函式會從 *piport* 得到它們的輸入。在下面的範例，假設檔案 *test.data* 有它的第一行被輸入如下：

#This is the data for test1

```
p = infile("test1.data")   ⇨   port:"test1.data"
getc(p)                    ⇨   \#
getc(p)                    ⇨   T
getc(p)                    ⇨   h
```

4.4 從字串讀入特定用途之格式

除了可以從終端機或文字檔去接受輸入資料， SKILL 也可以直接地從字串去得到它的輸入。有些應用是把程式存放在內部字串，透過分析這些字串，可將其轉成所需的 SKILL 內部表示方式。

因爲文法分析是一個浩大的工程 ，所以你應該避免對相同的字串去重覆地呼叫下面任何的函式。在多次使用它們之前，試著去轉換每個字串變成它內部的 SKILL 表示法是一個不錯的練習。

開啓一個字串供讀取（instring）

爲了讀取打開一個字串如同爲了輸入檔案內資料而打開檔案一樣。 傳回的是可以用檢讀取字串一個輸入埠。

```
s = "Hello World!"              ⇨   "Hello World!"
p = instring(s)                 ⇨   port:"*string*"
fscanf(p "%s %s" a b)?2
a                               ⇨   "Hello"
b                               ⇨   "World!"
close(p)                        ⇨   t
```

注意
當你執行完時，記得要關閉埠。

5. 系統相關函式

SKILL 提供各式各樣的函式來與系統環境進行互動。

5.1 執行 UNIX 命令

從 SKILL 的內部，你可以執行個人的 UNIX 命令或引用 *sh* 或 *csh* 等 UNIX 的 *shell*。

啓動 UNIX 的 Bourne-shell（sh,shell）

啓動 UNIX 的 Bourne-shell 子行程來執行一個 UNIX 的命令字串，如果呼叫 *sh* 函式而沒給參數，等於叫出一個交談式的 shell 環境，並提示你（們）輸入命令（僅非圖形的應用程式可用的）。

sh（"rm /tmp/junk"）

如果這成功移走把廢棄的文件從暫存的目錄移走，並傳回 t。

啓動 UNIX 的 C-shell（csh）

啓動 UNIX 的 C-shell 子行程來執行一個 UNIX 的命令字串，與 *sh* 功能的動作相同，只是啓動的像是 C-chell（*csh*）而不是 *shell*（*sh*）。

csh（"mkdir ~/tmp"）

在你的根目錄下，建立 tmp 的子目錄。

5.2 系統環境

下面的功能找到系統環境和與目前時間比較 ， 取得到你使用的軟體版本碼，和決定一個 Unix 作業系統環境變數的值。

得到目前時間（getCurrentTime）
getCurrentTime 以字串型式傳回目前時間。字串的格式是月　天　小時：分鐘：秒　年。

getCurrentTime( ) ⇨ “Jan 26 18:15:18 1993”

比較時間（compareTime）

compareTime 比較兩字串的參數 ，顯示日曆時間。字串的格式是月天小時：分鐘：秒 年。這些單位是鐘。

compareTime（”Apr 8 4:21:39 1991” “Apr 16 3:24:36 1991”）

⇨ -687777.

687,777 秒在兩個日期之間發生了。爲第幾秒鐘中的正數，最近的日期需要是第一個參數

compareTime（”Apr 16 3:24:36 1991” “Apr 16 3:14:36 1991”）

⇨ 600。

600 秒（10 分鐘）在兩個日期之間發生（出現）了

得到現在 CADENCE 軟體版本的號碼（getVersion）

傳回你現在所使用軟體的版本號碼。

getVersion（ ）

⇨ “cds3 version 4.2.2 Fri Jan 26 20:40:28 PST 1993”

得到 UNIX 環境變數的值（getShellEnvVar）

如果它已經設定，*getShellEnvVar* 傳回一個 Unix 作業系統環境變量的值。

getShellEnvVar（”SHELL”） ⇨ “/bin/csh”

傳回 SHELL 環境變數現在的值。

設定一個 Unix 作業系統環境變數的值（setShellEnvVar）

setShellEnvVar 設定 UNIX 環境變數的值爲另一個新的值。

setShellEnvVar（”PWD=/tmp”） ⇨ t

設定根工作目錄到目錄 /tmp 。

setShellEnvVar（”PWD”） ⇨ “/tmp”

設定根工作目錄。

8

進階操作

概要：

背景觀念

在第開始的第 27 頁已經介紹你如何建構 Cadence SKILL 語言目錄。這個節將教你更多建構目錄的細節。

目錄如何在虛擬記憶體幫助你更了解 SKILL 函數，如 *car* 和 *cdr* ，如此在建構大型的目錄之後會更有效率。

在虛擬記憶體目錄如何被存取

目錄和符號事實上是記憶體指標在處理，當你分配目錄到一個變數，而這一個內部變數是一個指向目錄頂端的變數，當一個目錄被函數分爲幾個部分，如 *car* 和 *cdr* ，只有指標指向不同的部分而沒有新的目錄被建立。

SKILL 平定你的指標問題在如何展示目錄和符號，一般來說，當 SKILL 陳列一個資料型態，會使用一個獨特語法去平定不恰當的細節和不可缺的資料重點。這一個語法有牽涉到目錄和符號資料型態。代替展示記憶體位址，SKILL 陳列有

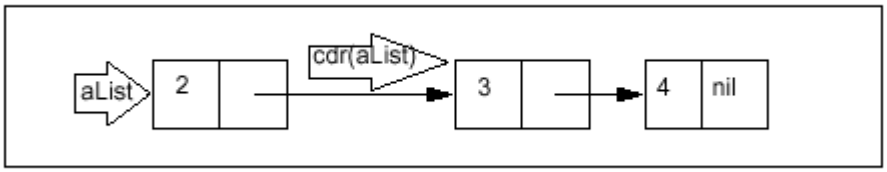
- ◆目錄周圍的元素
- ◆符號名稱

一個 SKILL 目錄就一個目錄組織

SKILL 描繪一個目錄，一個目錄組織佔兩個位置在虛擬記憶體。

- ◆ 第一個位置握住參考的第一個元素
- ◆ 第二個位址握住參考目錄的尾端，或其它目錄的組織或 nil

表示 *aList = '(2 3 4)*分配在下列三個目錄組織。



car 函數傳回第一個位置的值

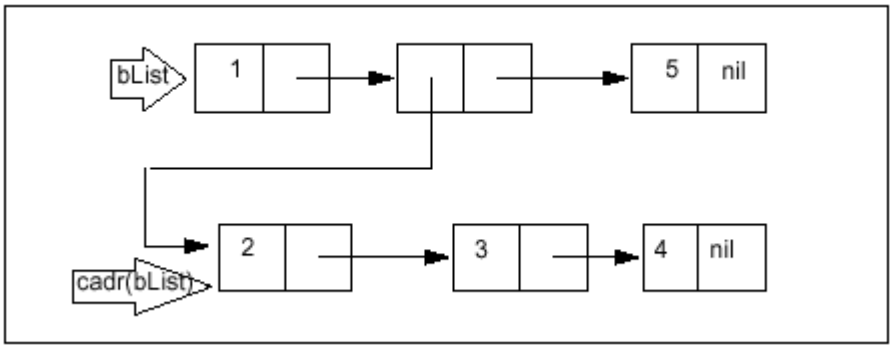
car (aList) => 2

cdr 函數傳回第二個位置的值

car (aList) => (3 4)

目錄包含附目錄

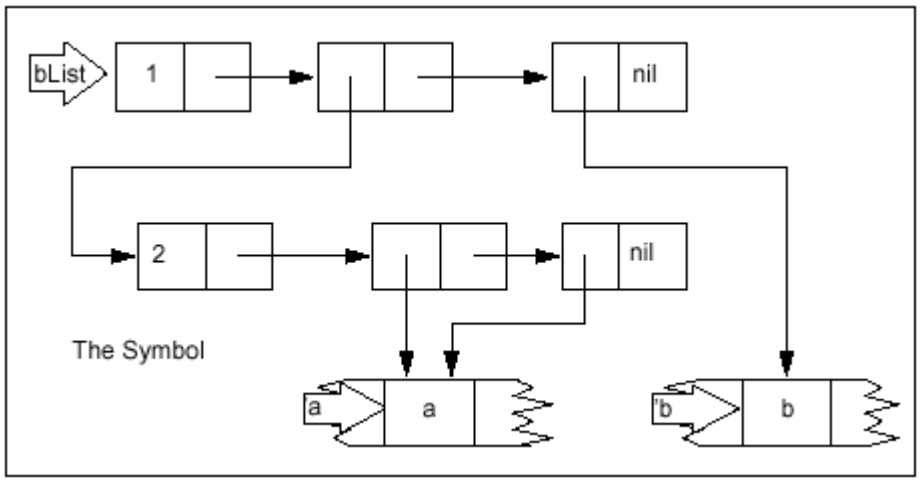
表示 *bList = '(1(2 3 4)5)*分配在下列三個目錄組織。



cadr(bList) =>(2 3 4)

目錄包含符號

表示 *bList = '(1(2 a a)b)*分配在下列三個目錄組織。



內部表示 *'a* 在符號表裡回傳指標到號 *a*

破壞性 vs 非破壞性操作

非破壞性操作

非破壞性操作的修改是有關任何分配一個目錄的複製。原始目錄沒有改變，更新任何變數是你的責任。

如一般的操作較易於會改變原始目錄的破壞性操作。而非破壞性會改變可能較費時的原始目錄。

破壞性操作

破壞性目錄修改是有關 *car* 或 *cdr* 兩者之間的目錄，破壞性修改函數不須要去建一個新的目錄架構，因此他們比非破壞性修改函數還快。

依據操作，任何變數有關的原始目錄會受到影響。很多微妙的問題出現當這些函數被使用而沒有經過理解。

警告

你應該只使用破壞性修改函數來描述而在這一個章節裡有非常好的 SKILL 語言理解力。

運作目錄總結

以下表格爲這一章的目錄操作總結。使用破壞性修改函數有好的地方。

操作清單

操作	函數	非破壞	破壞
改變目錄組織	rplaca , rplacd		x
存取目錄	nthelem , nthcdr , last	x	

操作清單

操作	函數	非破壞	破壞
建立目錄	cons,ncons,xcons	x	
	append1		
	tconc,cconc,lconc		
改編目錄	reverse	x	
	sort,sortcar		x
元素調動	remove,remq	x	
	remd,remdq		x
搜索目錄	member,memq	x	
	exists		
濾出目錄	setoff	x	
替代	subst	x	
穿越	mapc,map,mapcar,	x	
	maplist,mapcan		

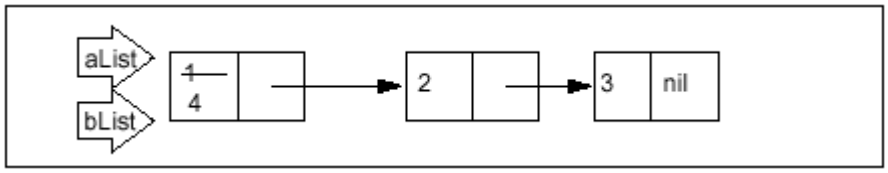
改變目錄組織

大多數基礎的破壞性操作有關於修改一個目錄組織。你能改變 *car* 或 *cdr* 組織中的其中一個。*rplaca* 和 *rplacd* 是破壞性操作運算子。

rplaca 函數

使用 *rplaca* 函數去把第一個目錄元素放回。

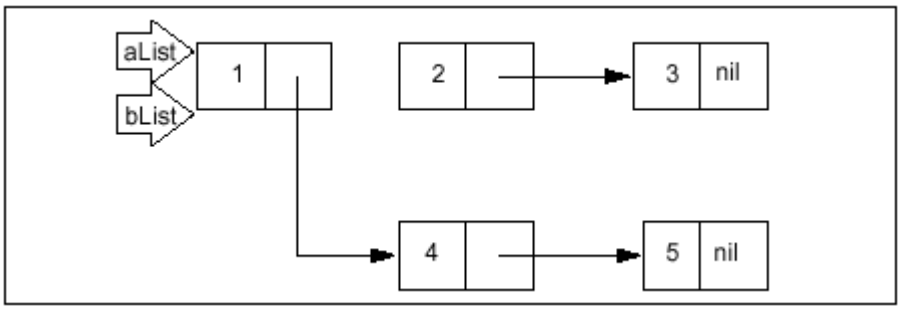
```
aList = '( 1 2 3) => ( 1 2 3 )
bList = rplaca( aList 4 ) => ( 4 2 3 )
aList => ( 4 2 3 )
eq( aList bList ) => t
```



rplacd 函數

使用 *rplacd* 函數還原目錄的尾端

```
aList = '( 1 2 3 ) => ( 1 2 3 )
bList = rplacd( aList '( 4 5 ) ) => ( 1 4 5 )
aList => ( 1 4 5 )
eq( aList bList ) => t
```



注意 *rplacd* 函數有要求修改回傳一個目錄，一個重點要記得破壞性操作所修改的目錄是在最初目錄虛擬記憶體的相同目錄。要證明這一個事實，請使用 *eq* 函數，他可以回傳 *t* 先在虛擬記憶體比較兩者相同的物件。

存取目錄

下列函數 cdr 和 nth 是既方便變動又有延展性的函數，介紹在第 27 頁 Getting Started

從目錄中選取索引元素

nthelem 回傳一個目錄索引元素，假有一個基礎索引。因此 n*thelem(1 l_list)* 和 *car(l_list)*相同。

```
nthelem( 1 '( a b c ) ) => a
z = '( 1 2 3 )
nthelem( 2 z ) => 2
```

請求 cdr 給一個次數數值到目錄裡

你提供了重複計算和目錄的元素。

```
nthcdr( 3 '( a b c d )) => (d)
```

```
z = '( 1 2 3 )
nthcdr( 2 z ) => ( 3 )
```

得到目錄最後的組織(last)

last 回傳目錄最後的組織。最後目錄的組織 *car* 是目錄最後的元素。

而 *cdr* 最後的組織爲 nil。

```
last( '(a b c) ) => (c)
z = '( 1 2 3 )
last( z ) => (3)
```

有效率的建構目錄

爲了有效率的建構目錄,你必須要了解如何建構目錄。使用一個函數像 *append* 去找尋目錄的尾端，這樣的慢爲大目錄所不能接受地。

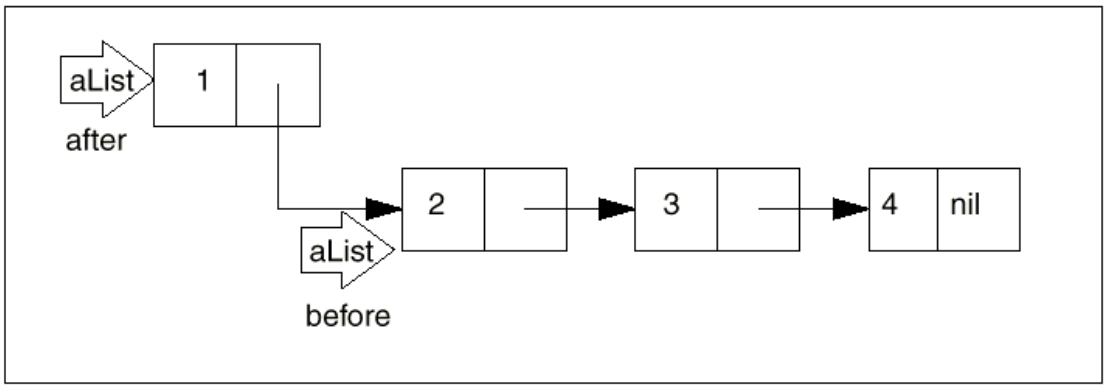
從目錄前加入元素(cons,xcons)

如果順序不重要，可以很容易的建構一個目錄，只要一直呼叫 *cons* 從目錄的頂端加入新的元素。

cons 函數會分配一個新的目錄組織來構成兩個記憶體位址。存取的第一個比較在第一個位址而存取的第二個比較在第二個位址。

```
aList = '( 2 3 4 )
aList = cons( 1 aList ) => (1 2 3 4 )
```

分配下列



xcons 和 *cons* 函數接受相同的比較，但是必須在相反的次序裡。*xcons* 加入元素到目錄的開始，因此會是 nil。

```
xcons( '( b c ) 'a ) => ( a b c )
```

建立一個有給元素的目錄(ncons)

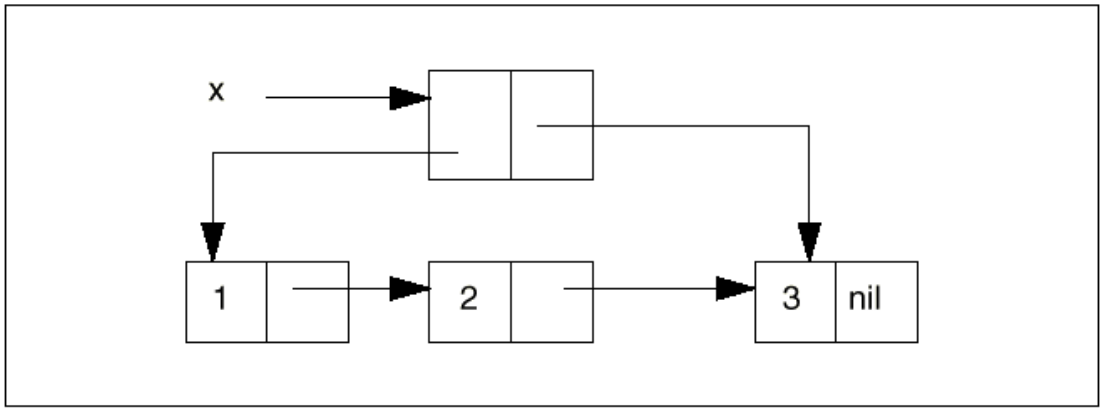
ncons 建立一個元素空目錄的開始。*cons(g_element nil)* 是相等的。

```
ncons( 'a ) => ( a ) ;;; equivalent to cons( 'a nil )
z = '( 1 2 3 )
ncons( z ) => ( ( 1 2 3 ) ) ;;; equivalent to cons( z nil )
```

加入元素到目錄的尾端(tconc)

因爲目錄在同一方向是單一連結，找尋典型的目錄結尾必須要通過每一個目錄組織，這能完全緩慢在一個長的目錄所包含的組織。如此長的穿越會出現一個問題，就是當你要去建立一個目錄加入到目錄的尾端。如果目錄必須建立在加入新的元素在目錄的尾端，最好的方法就是使用 *tconc* 。

tconc 函數創造一個目錄組織(在一個已知的架構中)而 *car* 指標指向目錄的頂端並且 *cdr* 指標指向目錄的元素的尾端。



tconc 架構允許後來被呼叫的 *tconc* 可以立刻找到目錄的尾端不必經過全部的目錄。依這個理由，呼叫 *tconc* 一次去初使化一個特別的目錄組織並且通過這特別的目錄組織到後來被呼叫。最後，爲了要得到一個實際的值你必須要建立，請使用 *car* 這個特別的目錄組織典型的步驟要使用 *tconc* 以下的方法：

1. 創造 *tconc* 架構呼叫 *tconc* 爲 nil 當第一個比較和第一個開始建立目錄的元素做第二次的比較。
   ```
   x = tconc(nil 1 )
   ```

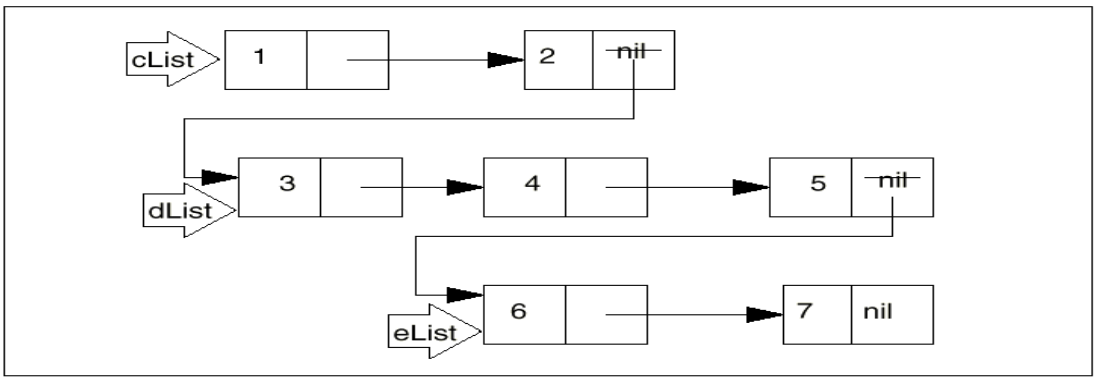
2. 再一次的呼叫 *tconc* 當有其它的元素要被加入到目錄的尾端，這一次就要給 *tconc* 一個架構在第一個比較的 *tconc*。不須要分配的回傳值，因爲 *tconc* 修改了 *tconc* 的架構，例如：
   ```
   tconc(x 2 ), tconc(x 3 ) ...
   ```

3. 目錄建立之後，取用 *tconc* 的 *car* 架構去得到一個實際目錄開始建立的值。如：
   ```
   x = car(x)
   ```

附加目錄

nconc 函數

使用 *nconc* 函數是爲了快速地附加目錄破壞性。*nconc* 函數花了二或更多的目錄在比較上。只有最後一個比較目錄是不被改變的。




```
cList = '( 1 2 )
dList = '( 3 4 5 )
eList = '( 6 7 )
nconc( cList dList eList ) => ( 1 2 3 4 5 6 7 )
cList => ( 1 2 3 4 5 6 7 )
dList => ( 3 4 5 6 7 )
eList => ( 6 7 )
```

使用 *apply* 函數和 *nconc* 函數去附加一個目錄中的目錄。

```
apply( 'nconc '( ( 1 2 ) ( 3 4 5 ) ( 6 7 )) )
=> (1 2 3 4 5 6 7 )
```

lconc 函數

lconc 使用 *tconc* 架構是爲了要有效率的接合目錄結尾中的目錄。而 *tconc* 架構必須用 *tconc* 函數來創造。請看以下的例子。

```
x = tconc(nil 1) ; x is initialized ((1) 1)
lconc(x '(2 3 4)) ; x is now ((1 2 3 4) 4)
lconc(x nil) ; Nothing is added to x.
lconc(x '(5)) ; x is now ((1 2 3 4 5) 5)
x = car( x ) ; x is now (1 2 3 4 5)
```

改編目錄

SKILL 提供了改編目錄的許多的函數。有時候最有效率的方法是去建立一個

目錄是反相的或是序列的之後你建立增值有 *cons* 函數。這些函數改變了你的目錄最高元素的順序。

反相目錄

以下是非破壞性的操作。

反相目錄元素的次序(reverse)

reverse 回傳一個目錄高等級的元素在反相次序裡。

```
aList = '( 1 2 3 )
aList = reverse( aList ) => ( 3 2 1 )
anotherList = '( 1 2 ( 3 4 5 ) 6 )
reverse( anotherList ) => ( 6 ( 3 4 5 ) 2 1 )
```

雖然 *reverseanotherList)* 回傳目錄在反相次序裡，*anotherList* 的值，初始目錄，沒有修改。更新任何的變數是你的責任

```
anotherList => ( 1 2 ( 3 4 5 ) 6 )
```

排列目錄

下列函數是有好處的當你必須排列目錄依據各種不同標準和在目錄裡的位址元素。他們是破壞性操作。

Sort 函數

語法說明 *sort* 函數是

```
sort( l_data u_comparefn ) => l_result
```

sort 排列一個物件的目錄(l_data)是要依據 sort 函數(u_comparefn)由你提供。u_comparefn(g_x g_y)回傳 non- nil 如果 g_x 能在 g_y 之前和 nil 則 g_y 在 g_x 之前，如果 u_comparefn 是 nil，則次序被使用。這是基本的遞回演算法。

```
sort( '(4 3 2 1) 'lessp ) => (1 2 3 4)
sort( '(d b c a) 'alphalessp) => (a b c d)
```

sortcar 函數

sortcar 相似於 *sort* 除此之外在目錄被使用排列函數比較的每一元素的 *car*

```
sortcar( '((4 four) (3 three) (2 two)) 'lessp )
=> ((2 two) (3 three) (4 four)
sortcar( '((d 4) (b 2) (c 3) (a 1)) nil )
=> ((a 1) (b 2) (c 3) (d 4))
```

目錄被攸改並且沒有新的貯藏被分配。而指標事先指示到目錄不會在排列目錄的頂端指示。

找尋目錄

SKILL 提供了許多令人滿意的標準元素的位址的函數。基本的標準是相等的，SKILL 能在兩者之間取得一個相等的值或是在記憶體位址取得相等的值。

member 函數

member 是一個簡單又詳細的函數在第 27 頁的 Getting Started。它使用了 *equal* 函數在基本的找尋一個高等級元素。

- 回傳 *nil* 如果元素不是 *equal* 在任何高等級的元素
- 回傳目錄的第一個結尾目錄的結尾開始是元素

包含一些例子

```
member( 3 '( 2 3 4 3 5 )) => (3 4 3 5)
member( 6 '( 2 3 4 3 5 )) => nil
```

member 函數類似 *cdr* 函數，內部的回傳值是一個指標指向一個目錄組織。目錄組織的 cdr 相等於元素。你能使用 member 函數有 *rplaca* 函數去代替一個元素爲第一個高等級的另一個現像。例如

```
rplaca(
member( 3 '( 2 3 4 3 5 ))
6 )
```

SKILL 提供了一個非破壞性的操作 *subst* 函數。請看第 191 頁 Substituting Elements

memq 函數

memq 函數和 *member* 函數是相同的除了使用 *eq* 函數去找到元素。因爲 *eq* 函數是更有較率於 *equal* 函數，使用 *memq* 無論什麼時候可能是基本的自然資料。例

如，如果目錄只去尋找包含符號，因此使用 *memq* 函數是更有效率於 *member* 函數。

存在函數

存在函數能使用一個應用特別的函數去定位第一個目錄元素的現象。*exists* 函數概括 *member* 和 *memq* 函數，這能定位第一個目錄元素的現象。

```
exists( x '( 2 4 7 8 ) oddp( x ) ) => ( 7 8 )
exists( x '( 2 4 6 8 ) evenp( x ) ) => ( 2 4 6 8 )
exists( x '(1 2 3 4) (x > 1) )=> (2 3 4)
exists( x '(1 2 3 4) (x > 4) )=> nil
```

複製目錄

有時候去請求一個破壞性的操作去複製一個目錄比請求一個非破壞性的操作更有效率。第一決定高等級的元素是足夠的。

複製函數

copy 回傳一個目錄的複製。*copy* 只複製高等級的目錄組織。所有的低等級的物件依然是共享的。你應考慮在製做任何目錄複製之前使用一個破壞性修改目錄。

```
z = '(1 (2 3) 4) => (1 (2 3) 4)
x = copy(z) => (1 (2 3) 4)
equal(z x) => t
```

z 和 x 有相同的值。

```
eq(z x) => nil
```

z 和 x 是不相同的目錄。

複製等級制度的目錄

下列函數是一個遞回複製的目錄。

```
procedure( trDeepCopy( arg )
```

```
cond(
( !arg nil )
( listp( arg );;; argument is a list
cons(
trDeepCopy( car( arg ) )
trDeepCopy( cdr( arg ) )
)
)
( t arg ) ;;; return the argument itself
) ; cond
) ; procedure
```

濾出目錄

很多目錄能夠抽象的考慮到在做一個濾出目錄的複製。濾出能在任何函數接受單一的比較。如果濾出函數回傳 *non-nil*，在新的目錄元素是被包含的。如果濾出函數回傳 *nil*，則元素不包括新的目錄。

setof 函數

setof 讓濾出一個目錄高等級的元素的複製，包含了所有安全給定標準的元素。例如，下列有兩個接近的對比去計算兩個目錄的交叉。

- 下列第一個方法是處理使用 *cons* 函數

```
procedure( trIntersect( list1 list2 )
"Return the intersection of list1 & list2"
let( ( result )
foreach( element1 list1
when( member( element1 list2 )
result = cons( element1 result )
) ; when
) ; foreach
result
) ; let
) ; procedure
```

更好的效果在下列：

```
procedure( trIntersect( list1 list2 )
setof(
element list1
member( element list2 ) )
```

```
) ; procedure
```

標準是決定是否包含目錄的每一個元素。複製元素不被改變。

目錄的元素調動

SKILL 有很多的函數是用來移動所有高等級現象的元素。

移動函數

	非破壞性	破壞性
Uses equal	remove	remd
Uses eq	remq	remdq

非破壞性操作

移動函數

remove 回傳一個比較的複製有所有高等級元素相同爲的是要 SKILL 物件移動。*equal* 函數工具 = 操作，被使用在相同的測試。

```
aList = '( 1 2 3 4 5 )
remove( 3 aList ) => ( 1 2 4 5 )
aList => ( 1 2 3 4 5 )
```

做一個適當的分配是你的責任。

```
aList = remove( 3 aList )
```

元素能讓自己變爲一個目錄

```
remove( '( 1 2 ) '( 1 ( 1 2 ) 3 )) => ( 1 3 )
```

remq 函數

remq 回傳一個目錄比較的複製有高等級元素的相同爲了給 SKILL 物件移動。*eq*

函數被使用在相等的測試。這函數比 *remove* 函數快因爲 *eq* 相同的測試較快於 *equal* 相同測試。然而 *eq* 測試必須在已知的資料型態裡，如符號和目錄。*remq* 函數比較合適，例如，當處理一個目錄的符號。

```
remq( 'x '( a b x d f x g ) ) => ( a b d f g )
```

remq 函數不會工作在相同的表裡。

破壞性操作

remd 函數

remd 移動所有高等級相等元素爲了要給 SKILL 物件。*remd* 使 *equal* 比較。這是一個破壞性的移動。

```
remd( "x" '("a" "b" "x" "d" "f")) => ("a" "b" "d" "f")
```

remdq 函數

remdq 移動所有高等級元素是爲了從目錄的第一個比較。*remdq* 使用 *eq* 代替 *equal* 來比較。這是一個破壞性的移動。

```
remdq('x '(a b x d f x g)) => (a b d f g)
```

代替元素

subst 函數是一個非破壞性的操作。更新任何變數是你的責任。

替代物件爲另一個目錄的物件(subst)

subst 回傳替代結果這個新的物件(第一個比較)爲所有先前物件相同的現象(第二個比較)。

```
aList = '( a b c ) => ( a b c )
subst( 'a 'b aList ) => ( a a c )
anotherList = '( a b y ( d y ( e y )))
subst('x 'y anotherList ) => ( a b x ( d x ( e x )))
```

將濾出目錄改變成元素

很多目錄操作能被做爲模型爲一個濾出通過跟隨。例如，假設你有一個目錄和你要去建立一個奇數矩形目錄。

階段 1：濾出奇數到目錄

你可以使用 *setof* 函數

```
setof( x '( 1 2 3 4 5 6 ) oddp(x)) => ( 1 3 5 )
```

階段 2：矩形元素目錄

你可以一起使用 *mapcar* 函數和一個矩形比較函數：

```
mapcar(
lambda( (x) x*x ) ;; square my argument
'( 1 3 5 )
) => ( 1 9 25 )
```

或是使用 foreach 函數：

```
foreach( mapcar x '( 1 3 5 ) x*x ) => ( 1 9 25 )
```

trListOfSquares 函數總結

```
procedure( trListOfSquares( aList )
let( ( filteredList )
filteredList =
setof( element aList oddp( element ))
foreach( mapcar element filteredList
element * element
) ; foreach
) ; foreach
) ; procedure
trListOfSquares( '( 1 2 3 4 5 6 )) => ( 1 9 25 )
```

有效的目錄

屬性是一個有效的函數是一個有安全標準的單一 SKILL 物件。SKILL 提供很多

基本的屬性。(參考 SKILL 屬性在 135 頁和種類屬性在 18 頁。)屬性回傳爲 *t* 或 *nil*。

SKILL 提供這 *forall* 函數和 *exists* 函數所以你可以檢查是否所有的要素或一些元素在一個安全的目錄標準。有二個函數是一致的數量詞在數位的邏輯裡。

■ *forall* 代表的數位符號是∀

■ *exists* 代表的數位符號是∃

這標準是代表著 SKILL 表現用一個單一引數你可以發現到這首先的參數是 *forall* 或 *exists* 函數。

forall 函數

*forall*証明出一個語法在每一個元素列表中保持真的原狀。這*forall*函數可以被使用在確認語在每一個一對的關鍵／値在綜合目錄保持真的原狀。(參考Association Tables在 115 頁可得到進一步的細節。)

```
forall( x '( 2 4 6 8 ) evenp( x ) ) => t
forall( x '( 2 4 7 8 ) evenp( x ) ) => nil
```

exists 函數

exists 可以使用應用一特性測試函數去找出第一個發生的元素在基本列表裡在相等之上。這 *exists* 函數概括著 *member* 和 *memq* 函數，確定在一個元素基本的列表上首先發生的事。

```
exists( x '( 2 4 7 8 ) oddp( x ) ) => ( 7 8 )
exists( x '( 2 4 6 8 ) evenp( x ) ) => ( 2 4 6 8 )
exists( x '(1 2 3 4) (x > 1) )=> (2 3 4)
exists( x '(1 2 3 4) (x > 4) )=> nil
```

使用 Mapping 函數去反對列表

SKILL 提供一個非常強大的語法函數爲了反覆目錄。爲了歷史的原因，有被稱爲映射函數。這五個映射函數分別爲 *map*，*mapcar*，*mapcan*，和 *maplist*。
所有五個 *map* 函數有相同的參數。

■ 一個函數，必需產生一個單一的參數。

■ 一個列表。

使用 lambda 用在 map 函數

它時常合宜使用在 *lambda* 構成定義一個無名的函數被使用當作首先的參數。例如:

```
mapcar(
lambda( ( x ) list( x x**2 )) ;;; return pair of x x**x
'(0 1 2 3 ))
=> ( (0 0) (1 1) (2 4) (3 9) )
```

參考Syntax Functions for Defining函數在 78 頁爲了更進一步的細節在lambda構思。

使用 map 函數在 foreach 函數

二擇一，你可以使用每一個映射函數當作 *foreach* 函數的一個選擇，時常做用 *foreach* 函數可獲得更多可瞭解的碼。

例如，這隨著而來的語法。

```
foreach( mapcar x '( 0 1 2 3 )
list( x x**2 ) ;;; build 2 element list of a x and x*x
)
=> ( (0 0) (1 1) (2 4) (3 9) )
mapcar(
lambda( ( x ) list( x x**2 )) ;;; return pair of x x**x
'(0 1 2 3 ))
```

這二個使用法之間的關係在映射函式是緊密的。當你使用 *foreach* 函數，SKLII 實際上合併了這個語法在 f*oreach* 主体的範圍內 *lambda* 函數當作說明下。這 *lambda* 函數被轉化成映射函數。例子，這隨著這個語法:

```
foreach( mapcar x aList
exp1()
exp2()
exp3()
)
mapcar(
lambda( ( x )
```

```
exp1()
exp2()
exp3()
)
aList
)
```

隨著特徵說明二者是方法去使用每一個映射函數。

mapc 函數

這 *mapc* 函數是缺乏映射函數在 *foreach* 巨集裡。當使用 *foreach* 函數

- 這 *foreach* 函數在引數列表中反覆每一個元素。
- 在每一個重覆，通用的元素 *foreach* 可用迴圈變數。
- 這 *foreach* 函數回傳原始的參數列表當作一個結果。

例子:

```
foreach( mapc x '(1 2 3 4 5)
println(x)
)
mapc(
lambda( ( x ) println( x ) )
'( 1 2 3 4 5 )
)
```

隨著這顯示:

```
1
2
3
4
5
```

這回傳值是(1 2 3 4 5)。

map 函數

這 *map* 函數是用在處理每一個列表組織，因爲使用 *cdr* 去消除這參數列表。當使

用這 *foreach* 函數

■ 這 *foreach* 函數反覆每一個列表組織的參數。

■ 在每一個重複，通用的列表組織 *foreach* 可用在變數的迴圈。

■ 這 *foreach* 函數迴傳原始的參數列表當作一個結果。

例子，假如你想要代替在個列表中使用查閱目錄的最高階層，隨著下面的例子:

```
trLookUpList = '(
( 1 "one" )
( 2 "two" )
( 3 "three" ))
```

使用 *map* 函數，你可得到接近每一個連續列表組織，隨著你去使用 *rplaca* 函數可以得到一個合適的代替。注意查詢列表是一個聯合列表所以你可以使用 *assoc* 函數去取回適當的代替。

```
assoc( 2 trLookUpList ) => ( 2 "two" )
assoc( 5 trLookUpList ) => nil
procedure( trTopLevelSubst( aList aLookUpList )
let( ( currentElement substValue )
foreach( map listCell aList
currentElement = car( listCell )
substValue = cadr(
assoc( currentElement aLookUpList ) )
when( substValue
rplaca( listCell substValue )
) ; when
) ; foreach
aList
) ; let
) ; procedure
trList = '( 1 4 5 3 3 )
trTopLevelSubst( trList trLookUpList ) =>
("one" 4 5 "three" "three")
```

mapcar 函數

這 *mapcar* 函數是用在爲了建立一個列表元素可以被取得一次-爲了-一次從這元素中一個最初的列表。當使用 *foreach* 函數

■ 這 *foreach* 函數反覆每一個元素的參數列表中。

■ 在每一個重複，通用的元素 *foreach* 可用在變數的迴圈。
■ 每一個重複產生一個數值結果。
■ 這些值被回傳在一個列表裡當作一個函數的結果。

例子:

```
foreach(mapcar x '(1 2 3 4 5)
println(x)
x*3
)
```

隨著這顯示:

```
1
2
3
4
5
```

這回傳值是(3 6 9 12 15)。

maplist 函數

這 *maplist* 函數使用在處理每一個列表組織因爲它使用在 *cdr* 去消除參數列表。像 *mapcar* 函數回傳列表的結果收集每一個重複值。當使用 *foreach* 函數

■ 這 *foreach* 函數反覆每一列表組織的參數列表。
■ 在每一個重複，通用的列表組織在 *foreach* 可用在迴圈變數裡。
■ 在每一個重複產生一個數值結果，和有數值被回傳在列表中當作 *foreach* 函數的結果。

例如，表慮代替例子說明 *map* 函數。附加這個代替，假如這函數在代替成功時必需顯示一個訊息獲得一個數值。

使用 *maplist* 代替 *map* 你可以建立一個列表在 1s 和 0s 反映這個代替執行時。使用 *apply* 函數用 *plus* 去把數值加起來去產生合適的數。

```
procedure( trTopLevelSubst( aList aLookUpList )
let( ( currentElement substValue substCountList )
substCountList = foreach( maplist listCell aList
currentElement = car( listCell )
```

```
substValue = cadr(
assoc( currentElement aLookUpList ) )
if( substValue
then
rplaca( listCell substValue )
1
else
0
) ; if
) ; foreach
printf( "There were %d substitutions\n"
apply( 'plus substCountList )
)
aList
) ; let
) ; procedure
trList = '( 1 4 5 3 3 )
trTopLevelSubst( trList trLookUpList ) =>
("one" 4 5 "three" "three")
There were 3 substitutions
```

mapcan 函數

像 *mapcar* 函數，這 *mapcan* 函數是用在建立一個被轉換的列表元素的原始列表。然而，代替收集這中間結果爲了一個列表當成 *mapcar* 在做，*mapcan* 添加他們。這中間結果必需被列表。注意這結果列表不需要被一對一的符合在原始列表裡。

這 *mapcan* 函數反覆每一個元素的參數列表。當使用 *foreach* 函數

- 在每一個重複，通用的元素在 *foreach* 可用在易變的迴圈裡，
- 每一個重複產生一個結果數值，這一個必需被列表。這些列表當時使連鎖被這破壞的修正函數 *nconc*，和這新的列表被回傳當作結果的函數當作一個全体的。

例如，變平坦一個列表的列表裡可以被容易做在

```
foreach( mapcan x '( ( 1 2 ) ( 3 4 5 ) ( 6 7 ) )
x
) => ( 1 2 3 4 5 6 7 )
```

```
mapcan(
lambda( ( x ) x )
'( ( 1 2 ) ( 3 4 5 ) ( 6 7 ) )
) => ( 1 2 3 4 5 6 7 )
```

另外的例子，隨著建立一個奇數的整數的方塊列表，在列表裡(1 2 3 4 5 6)。

```
foreach( mapcan x '( 1 2 3 4 5 6 )
when( oddp( x )
list( x )
)
) => ( 1 3 5 )
mapcan(
lambda( ( x ) when( oddp( x ) list( x )) )
'( 1 2 3 4 5 6 )
) => ( 1 3 5 )
```

概述這列表反對演算

這下面的表格概述如何在映射函數中工作。這項目函數參考這函數你傳遞映射函數。每一個表格記下一個映射函數行爲依照行和列方向。在一個例子，沒有如此的映射函數。

映射函數的摘要

映射函數回傳值	函數被連續的元素通過	函數被繼承列表組織通過
忽視這每一個重複的結果。 回傳這原始列表。	mapc (隱含值選擇)	map
收集每一個重複。 回傳這列表的結果。	mapcar	maplist
收集每一個重複的結果。 使用 nconc 去添加每一個 重複的結果。	mapcan	沒有這樣的函數

列出反對實列學習

這大部份有用的映射函數是 *mapc*，*mapcar*，和 *mapcan*。一個容易了解的函數是簡單化大部份是列表管理演算在 SKILL 裡。

較便利列表用一行

假定你需要去計算這長度在每個串列中在列表裡的串列以便這個長度被回傳在新的列表。就是這個函數應該執行隨著這變換:

```
( "sam" "francis" "nick" ) => ( 3 7 4 )
```

有一些不熟悉的 *mapcar* 選擇 *foreach* 可能的碼這隨著

```
procedure( string_lengths( stringList )
let( (result)
foreach( element stringList
result = cons( strlen(element) result)
) ; foreach
reverse( result )
) ; let
) ; procedure
```

使用這 *mapcar* 函數允許這個碼去被寫

```
procedure(string_lengths(stringList)
foreach(mapcar element stringList
strlen(element)
) ; foreach
) ; procedure
```

事實上，這 *foreach* 函數是當作一個巨集的工具，前者擴展一個映射函數，所以相同例子下可以被寫成立即地使用在映射函數隨著:

```
procedure(string_lengths(stringList)
mapcar( 'strlen stringList)
) ; procedure
```

做每個列表元素成爲一個子列表

你可以做這演算用任一個版本在 *trMakeSublists* 函數。

```
procedure( trMakeSublists( aList )
"return a list with every element of aList as a sublist"
foreach( mapcar element aList
```

```
list( element )
) ; foreach
) ; procedure
procedure( trMakeSublists( aList )
"return a list with every element of aList as a sublist"
mapcar(
'ncons ;;; return argument in a list
aList
)
) ; procedure
trMakeSublists( '(1 2 3)) => ( ( 1 ) ( 2 ) ( 3 ) )
```

使用 mapcan 爲了列表變平坦

這 *mapcan* 函數是用在當一個新的列表被取得從一個舊的表格，和每一個成員在舊的列表產生一個數字的元素在新的表格中。(注意那使用 *mapcar* 允許只有一對一的映射)。一個 *mapcan* 的應用在列表變平坦裡。假如我們有一個列表那包含列表的數目:

```
x = '( (1 2 3) (4) (5 6 7 8) () (9) )
```

這列表可以被變平坦使用這個程序:

```
procedure(flatten(numberList)
foreach( mapcan element numberList
element
) ; foreach
) ; procedure
flatten(x) => ( 1 2 3 4 5 6 7 8 9 )
```

這函數簡單地連鎖所有的子列表的參數，記得 *nconc* 是一個 *destructive* 修正函數，所以變數 x 不會較長保持有用的訊息，在之後的這個呼叫中。

保存數值 x，每一個子列表應該被拷貝在 *foreach* 函數的範圍內去產生一個新的子列表前者可以被無損害的修改:

```
procedure( flatten( numberList )
foreach( mapcan element numberList
copy(element)
) ; foreach
```

```
) ; procedure
x = '( (1 2 3) (4) (5 6 7 8) () (9) )
flatten(x) => ( 1 2 3 4 5 6 7 8 9 )
x => ((1 2 3) (4) (5 6 7 8) nil (9))
```

一個列表變平坦用很多階層

使用一個樣式的述語和一個遞歸的步驟，這程序可以被修改去變平坦一個列表用很多階層:

```
procedure(flatten(numberList)
foreach(mapcan element numberList
if( listp( element )
flatten(copy(element)) ;; then
ncons(element)
) ; if
) ; foreach
) ; procedure
x = '((1) ((2 (3) 4 ((5)) () 6) ((7 8 ()))) 9)
flatten(x) => (1 2 3 4 5 6 7 8 9)
x => ((1) ((2 (3) 4 ((5)) nil 6) ((7 8 nil))) 9)
```

這 *foreach* 的主体首先檢查這個通用的列表成員樣式。

- 如果這成員是一個列表，這結果是一個列表得到一個變平坦的拷貝。
- 如果這成員不是一個列表，它不可能被更進一步的變平坦，和結果是一個成員列表包含著這個成員。

記得當使用 *mapcan*，這結果的每一個重複必需被列出以便這結果可以被連鎖的使用在 *nconc*。

操作一個聯合列表

映射函數可以被非常強大的當使用在一起時。例如，假如有一個資料庫的名字和擴大的數字:

```
thePhoneDB = '(
("eric" 2345 6472 8857)
```

```
("sam" 4563 8857)
("julie" 7765 9097 5654 6653)
("francis")
("pat" 5527)
)
```

這資料庫被貯存在一個列表中用一個入口爲了每一個人。一個入口組成一個列表的人的名字隨著一個列表的擴大在那一些人可以被登入時。這列表的樣式被知道當作一個聯合的列表。一些人可以被登入在幾個擴大中，和一些人在所有中無一人。一個自動化計量系統已經被採用在接受只有名字-數字對:換言之，它需要資料在接著的型式:

```
(("eric" 2345)
("eric" 6472)
("eric" 8857)
("sam" 4563)
. . . . . .
```

怎樣可以資訊被轉換從一個樣式到另一個？每一個人(*person*)進入在最初的資料庫可以產生幾個入口在新的資料庫，所預 *mapcan* 必需被使用去反對這個數字。

從這個資訊，一個函式可以被寫成轉譯這資料庫:

```
procedure( translate( phoneDB )
let((name)
foreach(mapcan personEntry phoneDB
name = car( personEntry )
foreach( mapcar number cdr( personEntry )
list( name number )
) ; foreach
) ; foreach
) ; let
) ; procedure
```

展示這個工作，考慮這最內部的 *foreach* 基本的迴圈。這迴圈被叫爲了每一個人進入資料庫裡。假如這進入*(“sam” 4563 8857)*被處理。既然這樣，名字被設定成*”sam”*和 *foreach* 反覆這個列表的數字*(4563 8857)*。

因爲這重複是一個 *mapcar*，這結果的 *foreach* 是一個新的列表用一個進入爲了每一個數字在舊的列表中。每一次進入在新的列表裡是成對的名字-數字的構造語法列出(名字 數字)。因爲，這結果的 *foreach* 是這列表

```
(("sam" 4563) ("sam" 8857))
```

這最後的 *foreach* 連鎖這些列表的每對名字-數字:它工作完全像第一個例子的變平坦在事前做。注意不需要拷貝成員在之前連鎖的他們(因爲這例子變平坦了)因此他們必需被設計。

使用這存在函數去避免明確的列表反對

SKILL 提供一個大的數目的列表重複函數。這多數基本的是 *foreach* 函數，那些反對所有成員的列表，這反對是有用的，但是時常被需要去反對僅僅一部份的列表，去檢查爲了包含著情況。在這情況下，恰當的使用 *exists*，*forall*，和 *setof* 函數可以大大地改善你的碼。通常這個替代是可以花較少的效率。
考慮寫一個簡單的函數在反覆的所有整數列表裡和檢查是否有任何偶數整數。有幾個方法。

一個方法使用一個迴圈清楚地測試一個真假二元值變數被找到。

```
procedure( contains_even( list )
let( (found)
while( list && !found
when( evenp( car(list) )
found = t
)
list = cdr(list)
) /* end while */
/* Return whether found. */
found
) /* end let */
) /* end contains_pass */
contains_even( '(1 2 3 4 )) => t
contains_even( '(1 7 3 9 )) => nil
contains_even( nil ) => nil
```

另一個方法跳出 *while* 迴圈:

```
procedure(contains_even( list )
prog( ()
```

```
while(list
if( evenp( car(list) )
return(t)
)
list = cdr(list)
) /* end while */
return(nil)
) /* end prog */
) /* end contains_pass */
```

這方法可能表示的比較好因爲不需要區域變數。

第三個方法需要跳出 *foreach* 迴圈:

```
procedure(contains_even(list)
prog( ()
foreach(element list
if( evenp(element)
return(t)
)
) /* end foreach */
return(nil)
) /* end prog */
) /* end contains_pass */
```

但是這方法仍然需要 *prog* 去提供跳出 *foreach* 迴圈一次這成員有被找到。

一個比較簡單的方法是寫這個程序隨著:

```
procedure(contains_even(list)
when( exists( element list evenp( element ))
t
) /* when */
) /* end contains_pass */
```

這方法既不用 *prog* 也不用任何區域變數，也較直觀的。這 *exists* 函數限定一相配列表成員就被找到，所以這例子是正好有效率的在這關係裡因爲先前有一次了(必需使用 *return* 去完成這結果)。這結果的 *exists* 是 *nil* 如果沒有找到相配的登入。否則這結果是子列表的參數包含著相配成員當作它的項目。例如:

```
exists( x '( 1 2 3 4 ) evenp( x ) ) => ( 2 3 4 )
```

因爲 SKILL 考慮 *nil* 相等於假的和非-*nil* 相等於真的，這結果的 *exists* 可以被看作一個真假二元值是否合適的。注意如果這 *contains_even* 函數被限定去回傳任一個 *nil* 或非-*nil*(而不是 *t*)，它可以被更進一步的簡單化。

```
procedure( contains_even( list )
exists( element list evenp( element ))
) /* end contains_pass */
```

當所有其它的重複函數，沒有需要去宣告這迴圈變數爲了 *exists* 因爲這迴圈變數是區域函數和自動的宣告。

註釋列表的反對碼

上面的例子是有力的論証在映射函數。通常，無論什麼一個 *foreach* 迴圈被使用建立一個列表，一個映射函數通可以被替代使用。因爲映射函數收集所有的結果和應用一個單一列表建立演算，他們是較快於相等的重複函數和時常看起來簡潔的。

然而，這所有的映射函數看起來非常的相似，它是時常難以去完全的看出來在碼的部份，使用一個映射函數嘗試去做。爲了這個原因不論何時一個映射函數被使用，你應該寫一個完全的註解細節，對這函數被預期的回傳。

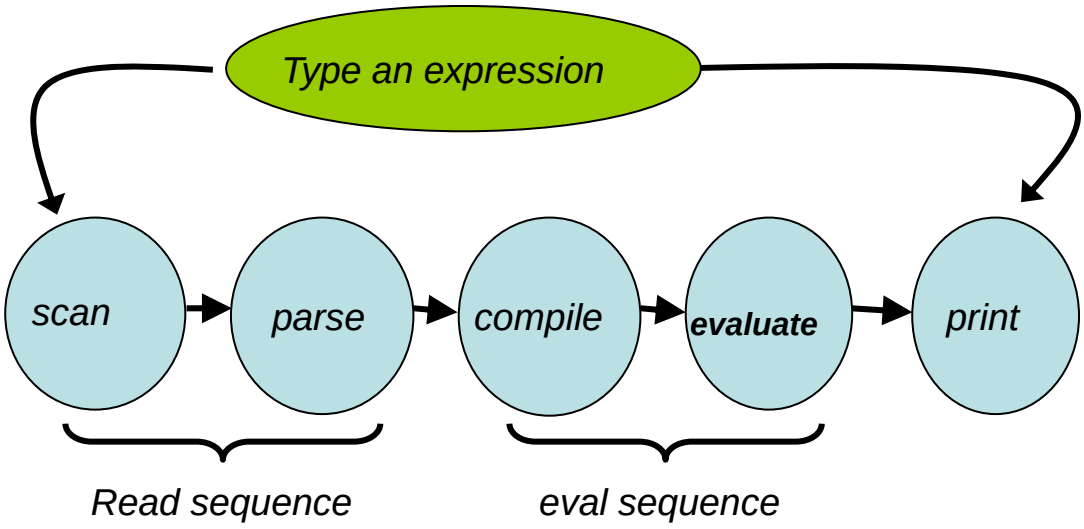
9

更進一步的要旨

Cadence SKILL 語言結構和執行

當你第一次接觸 Cadence SKILL 語言在一個應用並嘗試去表達求值，你會遇到已知在這個 *read-eval-print* 迴圈中。

表達你首先去嘗試”讀”和改變成可以求值的形式。這個數值去做在一個輸出的”eval”上從這”讀”。這結果的”eval”是一個 SKILL 資料數值，這是被列出的。這相同的連續是被反覆的，當其它的表示被開始時。



這”讀”在實例中隨著工作去執行。

- 這語法被分析，結果產生一個語法樹。
- 這語法樹在當時被組譯成一個主体的碼，當一個函數物件已知，做一個設定的訊息，何時實行，合適的結果影響著這個語法中。

這操作指南的產生是沒有特別的設算機結構。這一個程序是一個理論性的機器。通常，這設定的說明可以被認爲去在位元-碼或 p-碼。(p-碼是程序目標設定是相同的在這容易在 Pascal 編譯和提供這名詞的技術)。

這編譯者執行這個位元-碼產生一個值。就某種意義來說，這值的學習在軟体裡在硬体裡演算。這技術的理論說明設定被執行用一個工具寫進軟体。有幾個優點可以擴大語言的完成。

- 這技術提供恰當的基本解碼執行。
- 這技術提供較快的執行在直接來源解碼。
- 一旦 SKILL 碼被編譯成背景(參考傳送結果在 228 頁)這背景檔案是較快的負擔比原始來源碼和可攜帶的從一台機器到另一台。

如果這背景檔案被產生在一台機器上用在 A 結構從自動販賣機 X，它可以被拷貝在一個機器上用 B 結構從自動販賣機 Y 和 SKILL 將負擔這檔案不需要爲了再計算或轉譯。

求值

SKILL 提供函數運用在求值去執行一個 SKILL 的語句。你可以因此貯存程式當作資料來以後執行。你可以創造有力的修改或選擇數值函數定義和表達。

表達一個數值

eval 同意任何 SKILL 表達在一個參數。*eval* 數值一個參數和回傳它的值。

```
eval( '( plus 2 3 ) )=> 5
```

表達這數值 plus(2 3)。

```
x = 5
eval( 'x )=> 5
```

表達這符號 x 和回傳符號 x 的值

```
eval( list( 'max 2 1 ) ) => 2
```

表達這數值 max(2 1)。

獲得一個值的符號(symeval)

symeval 回傳數值的符號。*symeval* 是少許提高更有效率的比 *eval* 和可以被使用代替 *eval* 當你確定那參數被確實地求值符號。

```
x = 5
symeval( 'x )=> 5
y = 'unbound
symeval( 'y )=> unbound
```

Returns unbound if the symbol is unbound.

使用這個 *symeval* 函數用來求出在你紀錄名單內所遇到的符號之值。下列的 *foreach* 迴圈回傳例子來說， *aList* 的符號是根據它們的值做爲替換的動作。

```
a = 1
b = 2
aList = '( a b 3 4 )
anotherList = foreach( mapcar element aList
      if( symbolp( element )
            then symeval( element )
            else element
            ) ; if
      )
=> ( 1 2 3 4 )
```

將一個函數應用於一個證明的名單（apply）

apply 是一個拿來作爲兩個或很多的證明之函數。第一個證明必須是任一個符號用來象徵一個函數或函數物件的名字（ 在第212頁上有談論到宣告一個函數物件“lambda”）。 其餘證明允許被作爲如證明通用的函數。

呼叫這個 *apply* 函數可產生如第一個證明通用到其餘的證明。*apply* 是具有彈性化可作爲怎樣使證明通用到函數中。舉例說明，下面的所有呼叫都具有相同的效果， 允許 *plus* 到 1, 2, 3, 4 和 5 數字：

```
apply('plus '(1 2 3 4 5) )
=> 15
apply('plus 1 2 3 '(4 5))
=> 15
apply('plus 1 2 3 4 5 nil)
=> 15
```

到了最後證明的允許必定是一個記錄。如果功能是一個巨集的話，*apply* 計算的值只有一次，那是 *apply* 作爲擴充巨集和回傳擴充形式用。但是，不是用來計算原先擴充形式的值(如 *eval* 處理)。

```
apply('plus (list 1 2) )          ; 允許 plus 到紀錄中的 arguments。
=> 3
defmacro( sum (@rest nums) '(plus ,@nums)) ; 定義一個巨集。
=> sum
apply('sum '(sum 1 2))
```

```
=> (1 + 2)                              ; 回傳被延伸的巨集。
eval('(sum 1 2))

=> 3
```

函數物件

當你使用程序函數來定義在 SKILL 環境裡一個函數時，位元組編碼（byte-code）編譯器產生一個編碼區塊用來知道一個函數上的物件符號屬性和位置。

隨後，當 SKILL 在一個函數呼叫中遇到符號的時候，函數物件是被取回和求出執行指令之值。

函數物件可以被使用於分配指令和如通用證明函數的功能，好像有如的 *sort* 和 *mapcar 函數*一般。

SKILL 提供一些函數用來處理函數物件。

取回於一個符號的函數物件（getd）

你可以使用 *getd* 函數去取回函數物件，程序函數以符號作爲聯想。

舉例：

```
procedure( trAdd( x y )
    "Display a message and return the sum of x and y"
    printf( "Adding %d and %d ... %d \n" x y x+y )
    x+y
    )
=>  trAdd

getd( 'trAdd ) => funobj:0x1814bc0
```

如果沒有所謂的聯想函數物件話，*getd* 回傳爲空值。下表顯示出每個有可能回傳的值。

getd 函數

函數	回傳值	說明

trAdd	funobj:0x1814bc0	SKILL應用函數。
edit	t	SKILL 讀取保護函數（當載入是一個明文 或加密的檔案時）。
max	lambda:0xf6f25c	建立 lambda 函數。
breakpt	nlambda:0xf7a784	建立 nlambda 函數。

指定一個用來約束的函數(putd)

putd 函數用來約束函數物件爲一個符號。你可以根據設定的函數約束爲零值不用再定義一個函數。卻不可以使用 *putd* 來改變一個寫入保護函數的函數約束。從下面例子說明，你只能依下列方式拷貝一個函數來定義到其他符號：

```
putd( 'mySqrt getd( 'sqrt ))=> lambda:0x108b8
    指定函數 mySqrt 如 sqrt 相同的定義。
putd( 'newFun
    lambda( ( x y ) x + y )
    )
=> funobj:0x17e0b3c

newFun( 5 6 )
=> 11
```

一個函數定義指定 newFun 符號用來增加兩個常數。

宣告一個函數物件 (lambda)

在 SKILL 裡 *lambda* 這個字是被繼承於 Lisp 作爲微積分計算，爲 Lisp 上是一個基本的數學計算引擎。

lambda 函數是建立一個函數物件。這個論證到 *lambda* 函數是

- 有效的常數。
- SKILL 語法用來表達函數的個體。當函數物件是被通用於 *apply* 或 *funcall* 函數時，這些語法是被用來做求值用的。

不像程序函數般，*lambda* 函數不隨著任何獨特的符號聯想到函數物件。舉

例來說：

```
(lambda (x y) (sqrt (x*x + y*y)))
```

定義一個未命名的函數是要有能力去計算直角三角形的對角線邊之長度。

求出函數物件的值

未命名或匿名的函數在不同的情況裡是有效的。例如：*mapcar* 對應函數請求一個如第一個常數的函數。你可以通用於每一個符號或函數物件它本身。

```
mapcar( 'get_pname '( sin cos tan ))
=> ("sin" "cos" "tan")
mapcar( lambda(( x ) strlen( get_pname(x)) )

    '( sin cos tan ))
=> ( 3 3 3 )
```

註解：一個 *quote* 在一個 *lambda* 構成之前是不被需要的。事實上，一個 *quote* 在一個 *lambda* 構成之前是使用到比一個 *quote* 還緩慢的函數。因爲，在它呼叫常數之前需編譯很長一段時間。那是當編碼被載入時，*quote* 預防 *lambda* 常數是來至於所編譯到的一個函數物件。你可以在資料結構裡儲存函數物件，舉例說明：

```
var = (lambda (x y) x + y)
=> funobj:0x1eb038
```

這個函數物件儲存結果是可變性的 *var* 。函數物件是屬於第一級物件。那是可以使用函數物件剛好像一個任何其他類型的要求作爲通用如一個證明其他函數或指定一個可變的值。你也可以加上 *apply* 和 *funcall* 函數物件一起使用。舉例來說：

```
apply(var '(2 8))
=> 10
funcall(var 2 8)
=> 10
```

有如資料一般的有效儲存程式

無論何時如函數物件儲存 SKILL 程式是有可能代替文字字串的。函數物件是非常有效率的，因爲是呼叫到 *eval*、*errset*、*evalstring* 或 *errsetstring* 請求編譯和產生無相關分析文字字串。在其他方面上，最好不要引用 *lambda* 來表示曾經被編譯過。使用 *apply* 或 *funcall* 來作爲求值用。

轉換字串爲函數物件(stringToFunction)

使用 *stringToFunction* 時，一個轉換表示象徵一個字串成爲一個函數物件隨著零常數，例如：

```
f = stringToFunction("1+2") => funobj:0x220038
apply(f nil) => 3
```

一個轉換表示象徵一個紀錄清單，你可以隨同 *lambda* 和 *eval* 來構成一個清單。確信你可以說明任何的參數：

```
expr = '(x + y)
f = eval( '( lambda ( x y ) ,expr )) => funobj:0x33ab00
apply( f '( 5 6 ) ) => 11
```

你可以永遠參考如 *lambda* 的常數去構成它的表示，來避免最初一個多餘的呼

叫（*eval*）。

巨集

在 SKILL 裡的巨集是不同於在 C 中的巨集。

- 在 C 中，巨集本質上是爲對巨集的呼叫巨集本身的語句常數上之替換。

- 一個巨集功能允許你使用正常的 SKILL 函數，來呼叫文章結構配合你的應用需求。

巨集的益處

SKILL 巨集可以在各種不同的情形中被使用：

- 根據更換函數的呼叫具有 in-line 編碼是可以增加加速的。

- 爲易讀的擴充常數表示。

- 當做方便 wrapper 在已存在的函數之上。

注意：
留意使用巨集於 in-line 擴充於增加編碼的大小時，以便謹慎地去使用巨集和只有在哪一種情形中清楚將值加入於編碼內。

擴充巨集

當 SKILL 遇到一個巨集函數呼叫的時候，它是作爲立刻求出函數的呼叫之值。而且最後表達求出於當前的函數物件中所被編譯的值。這個程序是擴充巨集所呼叫的。擴充巨集是具有固有地的迴歸：一個巨集函數的本質可以表示其他的巨集包括它本身。

巨集應該是被定義在它是被當作參考之前。 這是最有效率處理巨集的方法。如果一個巨集是被參考之前已經被定義的話，如“*unknown*”的呼叫之編譯和執行中的求值擴充必定在巨集的效率會引起一個嚴重的問題。

重新定義巨集

如果你是在發展模式裡和你重新定義一個巨集的話，確定所有使用到的巨集編碼是被重新載入或重新定義的。 否則，無論巨集在何處擴充，原先的定義是被繼續使用的。

defmacro

在 SKILL 裡使用 *defmacro* 作爲定義一個巨集。

舉例說明，如果你想要檢查一個變數之值是一個字串之前呼叫 *printf* 或則是你不要寫入編碼內來作爲檢查每一個位置的完成，你可能考慮寫一個巨集：

```
(defmacro myPrintf (arg) ‘when((stringp ,arg)
printf( "%s" ,arg)))
```

雖然你可以看到巨集 *myPrintf* 使用 *backquote* 回傳一個擴充常數，在一個函數呼叫 *myPrintf* 的時間用來提換對 *myPrintf* 的呼叫所定義的。

mprocedure

mprocedure 函數是最早便利於以 *defmacro* 上爲根據的。避免使用*mprocedure* 在你開發中的新的編碼裡，使用 *defmacro* 作爲代替。當 SKILL 遇到一個 *mprocedure* 呼叫編譯中的原始碼時候，整體的紀錄清單表現出函數呼叫是直接地通用於*mprocedure*，用來限制單一正常的常數用。最後擴充所計算的結果在 *mprocedure* 內所編譯。

使用具有(')運算子的defmacro

這裡是一個實例的巨集，在 SKILL 的擴充巨集裡編譯時間效率是非常耀眼的。假定 *isMorning* 和 *isEvening* 是有被定義的。

```
(defmacro myGreeting (_arg)
    let((_res)
        cond( (isMorning()
                res= sprintf(nil "Good morning %s" _arg))
              (isEvening()
                _res= sprintf(nil "Good evening %s" _arg)))
        `println(,_res))
)
```

使用公用函數 *expandMacro* 來測試它的擴充，例如：

```
expandMacro(`myGreeting("Sue"))        ; 使用有關的定義
=> println("Good morning Sue")     ; 如果 isMorning 回傳 t
```

當被呼叫的時候， *myGreeting* 回傳一個 *println* 的宣告是依照合適的迎接 in-line 取代呼叫 *myGreeting*。因爲呼叫到 *sprintf* 內部裡的 *myGreeting* 是用來執

行外來的表示被回傳，作爲當編碼正在被編譯時或正在執行中所收到的訊息。這是如何讓 myGreeting 應該被重複寫入所有的歡應訊息反應時間作爲編碼正在執行中：

```
(defmacro myGreeting (_arg)
    ‘let((_res)
        cond( (isMorning()
                _res= sprintf(nil "Good morning %s" ,_arg))
              (isEvening()
                _res= sprintf(nil "Good evening %s" ,_arg)))
    println(_res))
)
```

如上述，當被編譯的時候，用整體 *let* 的擴充 in-line 取代巨集呼叫。

使用一個具有@rest的defmacro

下面的巨集例子顯示出一個機能如何可以根據開拓 in-line 擴充被有效率地實現。巨集的實現歸於 *letStar* （讀：*let-Star*）。它不同於是在有規律性的執行約束在序列的宣告區域變數，有如一個早期的宣告變數可以在擴充求出隨後變數宣告之值。這些在 *let* 裡是不安全去處理，舉例來說：

```
(letStar ((x 1) (y x+1) ...)
```

當約束 y 是被處理的時候，保證 x 是第一限制爲 1。

letStar 巨集

```
defmacro(letStar (decl @rest body)
            cond(( zerop(length(decl))
                    cons(‘progn body))
                ( t ‘let((,car(decl))
                        letStar(,cdr(decl),@body)))
            )
)
```

被迴歸定義。對於每個變數的宣告可以發源於 *let* 聲明，藉此保証所有的約束被持續地執行，下例說明：

```
procedure( foo()
        letStar(((x 1) (y x+1))
                        y+x))
```

延伸爲

```
procedure(foo()
```

```
        let(((x 1))
            let(((y x+1))
                progn( y+x ))))
```

使用具有 @key 的 defmacro

在下列例子說明自訂的語法作爲一個特別方法來建立一個紀錄清單，出至於一個原始紀錄是根據過濾器和變形器的運用。在紀錄裡去建立一個奇數的正方形清單。

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

你寫成

```
trForeach(
    ?element        x
    ?list               '( 0 1 2 3 4 5 6 7 8 9 )
    ?suchThat           oddp(x)
    ?collect        x*x
    ) => ( 1 9 25 49 81 )
```

而不是很多的編譯

```
foreach( mapcar x
    setof(x '(0 1 2 3 4 5 6 7 8 9) oddp(x))

    x * x
    ) => ( 1 9 25 49 81 )
```

如何建立 SKILL 的知識來實現一個 easy-to-maintain 巨集的請求，多去動態的使用 backquote（‘）、comma（,）和 comma-at（,@）運算子來表示。

下面跟隨 *trForeach* 定義。

```
defmacro( trForeach ( @key element list suchThat collect )
    ‘foreach( mapcar ,element
        setof( ,element ,list ,suchThat )
            ,collect
        ) ; foreach
) ; defmacro
```

變數

SKILL 使用符號作爲全域和區域兩者間的變數。在 SKILL 裡全域和區域的變數是來至 C 和 Pascal 兩個不同地方處理使用。

辭彙的Scoping

在 C 和 Pascal 中，一個程式可以提到一個區域變數只能確定本文程式的範圍所定義。這個範圍被變數的 *lexical scope* 所呼叫。例如：一個區域變數的 *lexical scope 是函數的本體*。在特色中

- 一個函數的外部，區域變數是難以接近的。
- 如果一個函數提到非區域變數的話，必定爲全域變數。

動態的Scoping

SKILL 不會全都依賴在 *lexical scope* 的規則。代替

- 一個符號的通用值是因你的應用在任何時間地點可以做存取的。
- 如果它是一個堆疊的話，SKILL 顯然管理一個符號的值跡象。
- 一個符號的通用值是簡單的堆疊頂端。
- 分配一個值到一個符號只能改變堆疊頂端。
- 每當控制的流程進入一個 *let* 或 *prog* 表示，系統會推入一個暫時的值到在區域變數紀錄裡的每一符號堆疊之值。
- 每當流程結束了 *let* 或 *prog* 表示程式，區域變數之前的值是被重新儲存的。

動態全域

在你的程式的執行期間，SKILL 程式語言無法區別全域和區域變數。

這個 *dynamically scoped* 變數術語是提到一個變數被使用如一個程序中的區

域變數，而且是在另一個程序中是全域。如此說來變數是最有影響的，因爲任何來至內部的 *let* 或 *prog* 表示函數是被呼叫可以改變 *let* 或 *prog* 的區域變數之值。舉例說明：

```
procedure( trOne()
    let( ( aGlobal )
        aGlobal = 5 ;;; 設定trOne的區域變數之值
        trTwo()
        aGlobal ;;; 回傳trOne的區域變數之值
        ) ;let
    ) ; procedure


procedure( trTwo()
    printf("\naGlobal: %L" aGlobal )
    aGlobal = 6
    printf("\naGlobal: %L\n" aGlobal )
    aGlobal
    ) ; procedure
```

trOne 函數使用 *let* 函數作爲定義 *aGlobal* 是一個區域變數。然而，*aGlobal* 暫時的值對任何的函數 *trOne* 呼叫是可以存取的。特別是 *trTwo* 函數可改變 *aGlobal* 。

這個改變不是直覺性預期的，而且可能引起的問題是非常難去分離的。SKILL List 會回報這個變數類型是個 " Error Global "。它是普遍地勸告使用者不應該依賴在變數約束的動態行爲上。

錯誤處理

SKILL 有一個健全的錯誤處理環境允許函數作失敗的執行和從使用者安全性的錯誤做補救。當一個錯誤是無法覆蓋的時候，你可以送一個錯誤信號至呼叫階層。這個錯誤是當時根據最先錯誤的主動捕捉所獲取的。內定的錯誤捕捉是SKILL頂端層次，找出不被你的自己錯誤捕捉所獲取到的所有錯誤。

errset 函數

errset 函數捕捉任何的錯誤所發出的信號在它本身執行期間。這個 *errset* 函數回傳一個值基於如何把錯誤送出信號。如果沒有錯誤被發出信號的話，在 *errset* 本身中的最後表示的值被計算於一個紀錄裡所回報。

```
errset( 1+2 )(3)
```

在下列例子中，沒有 *errset* 包裹器表示成

```
1+"text"
```

發出一個錯誤信號而且顯示訊息

```
*Error* plus: can't handle (1 + "text")
```

用來捕捉錯誤在一個 *errset* 裡覆蓋表示。捕捉到的錯誤引起 *errset 回報空值*。

```
errset( 1+"text" ) => nil
```

如果你使 t 通過如第二個常數的話，任何的錯誤訊息都被顯示出來。

```
errset( 1+"text" t ) => nil
*Error* plus: can't handle (1 + "text")
```

有關錯誤的資訊是被放在 *errset* 符號的 *errset* 屬性內。程式可能因此跟隨 *errset* 存取這些資訊。在 *errset* 常數之後決定了 *errset* 回報空值。

```
errset( 1+"text" ) => nil
errset.errset =>
("plus" 0 t nil
    ("*Error* plus: can't handle (1 + \"text\")"))
```

一起使用 err 和 errset

使用 *err* 函數來通用控制來至於發覺出在堆疊上結束 *errset* 的錯誤觀點。根據你的引述到的 *err* 函數，你可以控制回報 *errset* 的值。

如果這個錯誤根據一個 *errset* 所捕捉到的話，根據 *errset* 零值是被回報的。然而，如果一個可選擇的引述被給與，那值在一個紀錄中出於 *errset* 是被回報的，而且可以確認使用 *err* 發送訊號的錯誤。*err* 函數從來不回報。

```
procedure( trDivide( x )
    cond(
        ( !numberp( x ) err() )
        ( zerop( x ) err( 'trDivideByZero ) )
        ( t 1.0/x )
        )
    ) ; procedure
errset( trDivide( 5 ) )          => ( 0.2 )
errset( trDivide( 0 ) )          => (trDivideByZero)
errset( trDivide( "text" ) )     => nil
errset( err( 'ErrorType) )       => (ErrorType)
errset.errset                        => nil
```

error 函數

如果每一個被給與的話，然後有呼叫 *err* ，引起一個錯誤，*error* 會列印出錯誤訊息。最先的引述可能是格式化的字串，接著一直列印到最後。

```
error( "myFunc" "Bad List" )
        Prints *Error* myFunc: Bad List.
error( "bad args - %s %d %L" "name" 100 '(1 2 3) )
        Prints *Error* bad args - name 100 (1 2 3).
errset( error( "test" ) t)=> nil
        Prints *Error* test.
```

warn 函數

warn 佇列是一個警告訊息字串。之後一個函數回報到最高層次，所有的佇列警告訊息是被列印在 Command Interpreter Window（CIW）和系統揮霍的警告佇列。引述到的 *warn* 使用相同格式規格如： *sprintf*、*printf* 和 *fprintf*。

這個函數是使用於在一致的格式中列印 SKILL 警告錯誤。你也可以隨著一個稍後呼叫到的 *getWarn* 刪除一個訊息。

```
arg1 = 'fail
warn( "setSkillPath: first argument must be a string or list of strings - %s\n" arg1)
=> nil

*WARNING* setSkillPath: first argument must be a string or list of strings - fail
```

getWarn 函數

getWarn 衍生佇列大部分是最近排列的警告，出至於一個之前 *warn* 函數呼叫和回報警告的結果。

```
procedure( testWarn( @key ( dequeueWarn nil ) )
        warn("This is warning %d\n" 1 ) ;;; queue a warning
        warn("This is warning %d\n" 2 ) ;;; queue a warning
        warn("This is warning %d\n" 3 ) ;;; queue a warning
        when( dequeueWarn
            getWarn() ;;; return the most recently queued warning
            )
        ) ; procedure
```

如果 t 被獲准而且 *nil* 是如一個引述所附與所得到的警告的話，*testWarn* 函數會列印警告。

```
testWarn( ?dequeueWarn nil)
=> nil
*WARNING* This is warning 1
*WARNING* This is warning 2
*WARNING* This is warning 3
```

回報 *nil* 而且系統列印出所有佇列的警告。

```
testWarn( ?dequeueWarn t)
=> "This is warning 3\n"
```

WARNING This is warning 1
WARNING This is warning 2

回報最近的衍生佇列警告和系統列印出剩下的佇列警告。

頂端層次

當你執行 SKILL 或不是圖形介面應用建立在 SKILL 的頂端時，你是具有討論到 SKILL頂端層次，如輸入的讀取來置於終端機、求出表示之值和列印結果。如果一個錯誤是被遇到在表示的求值期間的話，通常都是經由回歸到頂端層次來控制。

當你在討論頂端層次的時候，任何完整的表示類型是被有實現性的求出值來（根據下面 Retry 鑰匙的類型去發出你的輸入結束信號）。下面的求出來的表示之值是被漂亮的列印出。

如果只有一個符號的名字在最頂端層次被分類， SKILL 檢查變數條件是被約束的。如此說來，變數的值是可被列印的。否則，符號是被拿來做一個函數的名字所呼叫的（不具有引述）。下列各個例子顯示出外部成對的括弧如何可以在頂端層次中被忽略掉。

```
if (ga > 1) 0 1
if( (a > 1) 0 1 )
loadi "file.ext"
loadi("file.ext")
exit                    ; 採取離開沒有變數的約束
exit()
```

假定有任何開放的括弧或任何二進制 *infix* 運算子沒有已經分配一個正確的運算子的話，它能夠安靜的等待你所終止的表示，這個缺失的頂端層次是使用 *lineread*。如果 SKILL 看似處理完事情之後你催促 Return 來改變你所擁有 mixtype 的事物的話，而且 SKILL 可因你的終止表示來做等待。

有時一個大括弧的類型（]) 是所有因你所需要的適當性終止輸入表示。假如mixtype 的事物當作輸入一個來至短路徑多重線路的話，你可以使用輸入按 Control+c 來作廢掉。你也可以按 Control+c 來中斷函數的執行。。

記憶體管理 (垃圾收集機制)

在 SKILL 裡所有記憶體分配和否定分配都是被自動化的管理。那是讓使用SKILL的開發者不用去記住未分配到的所使用結構。例如：當你建立一個陣列或一個 *defstruct* 和指定一個值到變數區域宣告程序的要求時，假定使用之後程序離開結構不是很長的話，記憶體管理會自動回收這些結構。事實上，回收結構是被隨後

循環再利用。在 SKILL 裡使用者程式是利用垃圾收集器來簡化紀錄所有使用者沒有關於儲存管理困擾的點。

分配者保持一個記憶體的池於每一個資料格式，而且分配者在請求上對於各式各樣的記憶池和回收結構是被回傳到池裡。當一個記憶池被用盡的時候，垃圾收集器 （GC）被觸發，這時回收處理沒有使用到記憶體。

垃圾收集器藉由追蹤所以未使用或爲釋放的記憶體來補足記憶池，而且重新產生有效的分配。如果垃圾收集器不能夠充份回收的記憶體的話，分配者啓動根據在執行中的決定因素來啓發式擴充記憶池。

垃圾收集器對 SKILL 使用者是透明的，並且對應用類型的使用者建立在 SKILL 之上。當垃圾收集器被觸發的時候，系統可能爲短暫階段的關掉，但是在大部份的情形下它不應該是引人注目的。然而，在 SKILL 應用中抑止記憶體，對於打算使用垃圾收集器下作爲抑止可能在很多情形的結果。

如何具有垃圾收集器的運作

垃圾收集器使用一個啓發式的程序作爲動態的決定，必須因於附加運作的記憶體應該被來至作業系統的分配。這個程序運作可在很多情況下，做爲不分時間和空間的相關應用合適的變化交易，你可能有時想要放棄內定的系統參數。

當一個應用已知道使用 SKILL 內部資料格式很多時，你可以計算記憶池所需量的期間和預先分配記憶池時間。這個分配協助簡化追蹤在一個垃圾收集器的號碼期間。然而，因爲根據垃圾收集器那額外狀況是只有在執行中單獨的百分比才作代表性的，導致 fine-tuning 不應該在很多系統值得處理的。

首先你需要根據使用 *gcsummary* 來分析你的記憶體慣例。這個函數列印一個記憶體分配在分解期間。可從下面輸出例子看出。一旦你有決定多少資料格式的要求於你需要的期間，你可以根據使用 *needNCells* 預先分配記憶體的資料格式（這個部分在最後有詳細描述）。

大致上，你不需要去調整記憶體的慣例。你應該首先使用記憶體寫照來做觀察 （在 Cadence SKILL Profiler in SKILL Development Help 有提到）是否你可以追蹤到哪裡的記憶體是被產生的和最先的交易。在這個部分使用記憶體調整技術描述有如是最後的手段。因爲所有的記憶體調整是全域性的，所以記得不可以只有調整記憶體於你的應用。申請不能僅僅調節記憶。 所有的其他應用執行在相同執行二位元是根據你的調整所影響的。

列印結論狀態

gcsummary 函數是列印一個記憶體分配和在通用 SKILL 執行裡垃圾收集器狀態的結論。

如何中斷結論的回報

行	含有
Type	資料格式名稱。
Size	各微量的大小代表在位元組理的資料格式。
Allocated	在記憶池裡位元組分配的全部號碼於資料格式。
Free	位元組的號碼是自由的和有效的分配。
Static	在靜態池裡記憶體分配不是隸屬於GC.。當文字的建立時記憶體是通常被產生的。當變數是做寫入保護時，那些文字是被移至到靜態池裡。
GC Count	GC的號碼週期觸發因爲這個資料格式因記憶池所被排放。

```
************* SUMMARY OF MEMORY ALLOCATION *************
Maximum Process Size (i.e., voMemoryUsed) = 3589448
Total Number of Bytes Allocated by IL = 2366720
Total Number of Static Bytes          = 1605632
-------------------------------------------------------------
Type       Size   Allocated     Free      Static   GC count
-------------------------------------------------------------
list         12      339968    42744     1191936          9
fixnum        8       36864    36104       12288          0
flonum       16        4096     2800       20480          0
string        8       90112    75008       32768          0
symbol       28           0        0      303104          0
binary       16           0        0        8192          0
port         60        8192     7680           0          0
array        16       20480     8288        8192          0
TOTALS       --      516096   188848     1576960          9
-------------------------------------------------------------
User Type (ID)              Allocated     Free    GC count
-------------------------------------------------------------
hiField              (20)       8192     7504           0
hiToggleItem         (21)       8192     7900           0
hiMenu               (22)       8192     7524           0



hiMenuItem           (23)       8192     5600           0
TOTALS                --       32768    28528           0
-------------------------------------------------------------
Bytes allocated for :
      arrays       = 38176
      strings      = 43912
      strings(perm)= 68708
      IL stack     = 49140
      (Internal)   = 12288
TOTAL GC COUNT 9
----- Summary of Symbol Table Statistics -----
Total Number of Symbols = 11201
Hash Buckets Occupied = 4116 out of 4499
Average chain length = 2.721331
```

手動分配空間

這個 *needNCells* 函數帶有組織計數和分配適當的頁數號碼到所容納的組織計算。使用者格式的名字可以被通用於字串和符號裡。然而，內部的格式像 *list* 或*fixnum*，必定要在符號裡所通用。舉例來說：

needNCells('list 1000)

保證在這系統裡總是會有1000紀錄組織是有效的。

離開 SKILL

當正常離開 SKILL 目錄時，可以從 Cadence 的圖形介面 CIW 選擇到 *Quit* 選單命令或是 Control+d 格式於執行中的非圖形模式來做離開。然而，你可以呼叫*exit* 函數做離開一個執行中的應用或不要明確的狀況編碼。實際上，在 SKILL 裡*Quit* 選單和 Control+d 兩者都是觸發呼叫到*exit*。

有時你可能像做處理確定清理完行爲之前離開 SKILL。根據所註冊的離開之前（*regExitBefore*）或（和）離開之後（*regExitAfter*）的函數，你可以去完成這些。一個 *regExitBefore* 函數是被呼叫之前 *exit* 所做的任何事，而且 *regExitAfter* 函數是呼叫之後 *exit* 所完成紀錄的工作和之前剛好回報作業系統的控制。這些使用者定義的離開函數不帶任何的引述。

爲了要有很多規則的控制，一個 *regExitBefore* 函數可以回報微量 *ignoreExitto* 關於完全離開呼叫。當一個 *exit* 被呼叫時，在反向成長的註冊裡最先所有註冊離開之前函數是被呼叫的。第一的所有註冊的出口－在功能是 calledin 登記的反面次序之前。如果它們哪個回報特別微量 *ignoreExit* 離開請求是被失敗，而且它會回傳 *nil* 來做清除的。

在呼叫 exit-before 函式之後，它做了一些記錄的工作，在它們的註冊中反方向的呼叫已註冊的 exit-after 函式，最後離開到作業系統。

對於 SKILL 較早版本的相容性，你仍然可以定義函式名稱 *exitbefore* 和 *exitafter* 作爲其中之一的離開函式，它們被視爲第一個註冊的離開函式(最後被呼叫)。

避免令人困惑的系統設定，不爲其他目的使用這些名稱。

13

有關 SKILL++和 SKILL

本章資訊：

- 簡介 Cadence SKILL++語言
- SKILL++到 IEEE 和 CFI 標準的相對組合
- 擴展語言環境
- 說明語言
- 對照變數 Scoping
- 對照符號使用
- 對照使用資料的函式
- SKILL++結束討論
- SKILL++環境

介紹 Cadence SKILL++ 語言

The Cadence-supplied Scheme 是用來作 the Cadence ® SKILL++ language 的東西。有兩個主要因素影響 Cadence 的決定提供支持 SKILL++ ，在 Cadence SKILL 的工程和標準的軟體環境之間。

什麼是 SKILL++?

SKILL++是第二種普通的擴充語言的名字來自 Cadence 的 CAD 工具，它包含簡易使用 好的接收 SKILL 環境在電力極高的需求程式語言 Scheme。爲了給使用者更多接受的習慣和擴充的發展平臺。

Lexical Scoping 和電源關閉

自系統帶來主要的電力是用作“lexical scoping”和功能隨著語彙關閉環境稱之爲關閉。Lexical Scoping 使可靠的和模件程式更簡單達成。因爲你有全部控制可獲得的使用和參考任何符碼而沒有錯誤關於突然的墮落肇因在同樣偏僻的地方。

關閉是給充滿電力稱號，這只存在進一步的語言程式中。他們進入一個個別的單位內部壓縮符碼和有關的資料，隨著全部輸出控制的介面中。很多現代的程式慣用語和範例，就像 message-passing 和物體隨著繼承，可以作爲使用關閉的優雅的手段。

環境就像第一課程科目

此外，SKILL++提供“環境“就像第一個課程科目。隨著終止和第一課環境，使用者可以簡單創造他們擁有的組件或口袋的系統。換句話說，在第一課程的環境下，使用者不再限制於單獨平坦的名稱空間模型使用靠 SKILL。每個人現在都可以簡單組織符碼進入名字空間的等級制度。

SKILL++ Object Layer

除了 Scheme 語義學之外，SKILL++包含一個物體階層 使詳盡的以物體爲目的方式之程式變爲可能。物體階層支援分類、總類功能、和單一繼承權。

SKILL and SKILL++ Work 的融合

因爲 SKILL++和 SKILL 在同樣的環境裡可以共存融洽。向後的一致性和 interoperability 將不再存在。全部已存在的 SKILL 符碼可以繼續執行而不會改變，而且全部或部分任何 SKILL 包裝可以移到 SKILL++.。SKILL++ and SKILL 的符碼發展可以明顯的叫做 互相和分享 同樣的資料架構。

SKILL 的背景

SKILL 是基於說不清的獨創叫做“Franz Lisp.” 。Franz Lisp.和其他全部風格不清楚是最後 superceded 依照 ANSI 標準爲了說不清叫做“Common Lisp.”。

語意學和 SKILL 的自然性使他它理想爲手寫和快速雛形。但是它缺少模式化和良好資料抽象，或者他的一般性寬廣，使它更難去要求現代的軟體工程原理是特別地爲了更大的努力，自從那時開始，SKILL 一直是用來寫很大的系統在 Cadence 工具環境之間。爲了這個 Cadence 選擇提供 Scheme 在 SKILL 的環境間。

方案(Scheme)的由來

方案(Scheme)是一種 Lisp-like 的語言起初地是在麻薩諾塞洲技術學院爲了教授電腦科學而被發展的並且是現在電腦科學和以前的電氣工程課程受歡迎的一種語言。Scheme 是一種現代的語言並受於工程師去發展聲音軟體系統。這有電機及電子工程師學會(IEEE)Scheme 的標準。Scheme 也是 CAD 主動性架構(CAD Framework Initiative (CFI))的選擇爲了擴充的語言基礎。

Scheme 被支在 SKILL 環境裡

Cadence 正提供主要的 Scheme 相關像 SKILL 的部分環境。這有很多好處:

- Scheme 中新寫的程式可以共存而且在呼叫現存程式中程序是被寫入在 SKILL 中而沒有。
- Scheme 和 SKILL 將共享執行時環境因此架構分派在 Scheme 程式中可以被忽略而沒有減少去 SKILL 程式和 visa-versa。
- 供應者和消費者可以選擇移到每一個獨立的 Scheme。例如，一個被支援的 Cadence-supplied 層面依然可以選擇在 SKILL 而當這些層面的使用者可以轉換去使用 Scheme 和 visa-versa。

IEEE 對 SKILL++的關係和 CFI 標準 Scheme

CFI 有選擇 IEEE 標準的 Scheme 像他們的 CAD 架構擴充語言過程基礎。因爲預期中的使用是像一個 CAD 工具擴充語言，必須做可嵌入的在大功用之中混合語言環境。CFI 釋放必需品給那些完全支援“call-with-current-continuation”和完全數字上的高塔（只有 C’s 長度相同的號碼和被請求是加倍的）。

爲了相同的理由，CFI 增加一些擴充的像是例外的處裡和函數爲了評估 Scheme 的密碼，好比詳述在外國的函數介面 APIs，他們所提出的建議。

SKILL++是隨著電機及電子工程師學會(IEEE) Scheme設計和CFI Scheme在記憶的順從。但是由於 SKILL 繼承和順從，他不全是順從和甚至是標準。接下來的章節描述在 SKILL++和 Scheme 語言的標準之不同。

語法的不同

SKILL++使用同樣相似的 SKILL 語法隨著限制和擴充。

限制

- 因爲大多數的特質在 SKILL++ 和 SKILL 中是被使用在插入符號。他們不能使用像常態性的名字來組成。然而，很多標準的 Scheme 功能和語法形式有系統的重新命名爲了在容易使用 SKILL++的句法之下，例如

  pair? ==> pairp

  list->vector ==> listToVector

  make-vector ==> makeVector

  set! ==> setq

  let* ==> letseq (for “sequential let”)

- 除了向量文字(e.g. #(1 2 3)),“#...語法是不鼓勵的，所以用't'代替#t, 'nil' 代替#f，而且用單獨的符號來描述不實的 l 文字諸如此類。

擴充

- 像是 SKILL； 但不像標準的 Scheme，SKILL++符號是敏感的事例。
- SKILL++繼承全部 SKILL 語法的特徵，像是寫入標記法、設定值非必須
- 是鍵盤參數、和很多充滿有效率的循環特殊形式（像是 *for* f*oreach, setof*）
- SKILL++碼可以定義巨集 macros 的使用 *mprocedure/defmacro* 像是使用
- 在 SKILL 裡已存在的 *macros* 定義。

語義的不同

SKILL++隨著限制和擴充採用標準的 Scheme 語義。

限制

- 原子零和空的陣列 list’()以及 *false* 值是相同的，標準的 Scheme 使用#f 來替代僅一個錯誤值而且對空的目錄像真的值一樣。
- 在 SKILL++中 cons cell 就像 SKILL's，即使，他們的 cdr 槽只能做 nil 或是其他 *cons cells*。在標準的 Scheme，一個 cons cells 的 cdr 槽可以維持任何數值。
- SKILL++的 *map* 函數和 SKILL 的 *map* 函數共用同樣的名字和實行； 但行爲不同於標準 Scheme 的地圖函數。要得到 Scheme 地圖函數的行爲，在

SKILL++使用 *mapcar*。

- SKILL++ 和 SKILL 的附帶條件是永遠不變的，所以這裡是不支援像 Scheme 的 *string-set!*函式。
- 但 *character* 形式是不支援的，像在 SKILL 一個特徵的符號可以像多個特徵一樣使用。
- 沒“eof”物件。*lineread* 函數在一個檔案結束後回到無。
- 繼承呼叫最佳化是正常的關閉（爲了更好的障礙排除和支援時光回朔）。因爲那裡有很多的循環構造繼承自 SKILL，這在 SKILL++程式上是正常的而不是個問題。

■ 擴充

- 作爲頭等的處理環境物件。*TheEnvironment* 函式可以習慣得到封閉的語彙環境，而且約束環境可以簡單做存取。這提供一個有效率的封裝工具。
- SKILL++繼承 SKILL 充效率的資料架構的調整，就像 *defstruct* 和聯想表格，好比全部的函數和很多特殊的 SKILL 形式。
- 支援易懂的交叉語言（SKILL <-> SKILL++）混合程式。

語法選項

- SKILL++採用和 SKILL 一樣的與法，命名爲 SKILL++程式可以寫在相近的嵌入語法和普通的表格處理程式句法。
- 假如語法對你來說不是問題要點而且你是舒服的在嵌入標記，那就繼續去使
- ■ 用它。
- 如果你關心 SKILL++的知識，你用程式建立嵌入語法將不再好用如果你
- 在 Scheme 環境寫程式（而沒有 SKILL），
- 然後 SKILL++程式使用表格處理程式語法。語法學在 SKILL++使用表格處理程式語法和標準 Scheme 之間的不同是：
- 在 SKILL++的鑑定裡你不能使用任何特殊的字元（像是+, -, /, *, %, !, $, &諸如此類），因爲大部分的字元是用做像嵌入操作的。
- 就像普通的慣例，Scheme 函數提供隨著驚嘆號(!)結束而沒有驚嘆號或者像同等於 SKILL 的功能。舉例來說，Scheme set! 是 SKILL++ *setq*。Scheme 函數使用“->”提供使用“To”就像功能名稱的一部分。舉例來說，“list->vector”變成
- *list To Vector*。見
- 對於 *name mappings* 的一個完整 SKILL 語言目錄參考附錄 A SKILL++ Equivalents 的表格。
- Scheme 的雙點在 SKILL++是不可用的。取而代之使用簡單的目錄表代替。
- 使用“=>”和“...”標示符號是被支援的。

順從與拒絕者(Compliance Disclaimer)
Scheme 像備有 Cadence 一樣將不會有充分的 IEEE 順從不同的理由。Scheme 像擴充的語言一樣是非組織的設計。所以 Scheme 的特徵不能安全使用連接系統在寫 C/C++是被忽略的。像是“call/cc”是非無效的終止字串。Cadence 放置在一個高價值的使充滿向後並立系統爲了 SKILL 程式和程序的介面。結果，空的名單‘*nil*’是一個 *true* 假二值真的在 Scheme 標準，反之 Cadence's SKILL 和 SKILL++對待無就像真假二值的假的。若沒有這對待‘*nil*’，存在的遷移 SKILL 程式 Scheme 寫在 SKILL 要改變將會需要很多存在的程序的介面。一般來說，SKILL++是爲了寫 Scheme 程式而用來支援表格處理程式句法和更多相近的 SKILL 嵌入句法的工具。就像爲遷移 SKILL 碼到 Scheme 提供一個順暢的路徑。

參考

爲了進一步閱讀 Scheme:
電腦程式的架構和翻譯, Harold Abelson, Gerald Sussman, Julie Sussman, McGraw Hill, 1985.

Scheme and the Art of Programming, G. Springer & D. Friedman, McGraw Hill, 1989.An Introduction to Scheme, J. Smith, Prentice Hall, 1988.
“Draft Standard for the Scheme Programming Language,” P1178/D5, October 1, 1990. IEEEworking paper.

擴充語言環境

Cadence 的擴充語言環境支援兩個統合的擴充語言，SKILL 和 SKILL++，在其中一個語言程式裡是不被拘束的。

- 資料共享
- 每一函數的 cell

你的應用可由寫在其中一個語言的來源碼構成，本章幫助你選擇語言使用你的軟體進展。

主要的存在 SKILL 的程式

你可以主要存在 SKILL 應用而不改變任何東西。

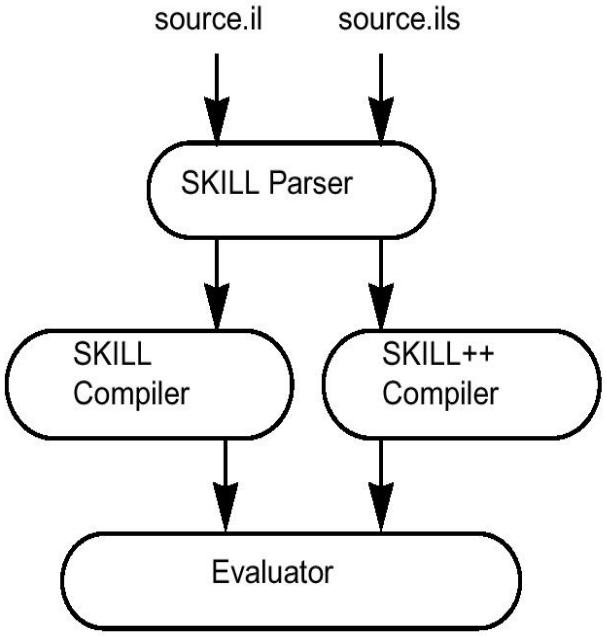
用 SKILL++隱藏私人函數和私人資料

在新應用方面使用 SKILL++使你隱藏私人函數和私人資料。你可以因此利用多次使用成分設計和裝備你的應用，如果你要發展新的應用，在必要時你應該認爲使用 SKILL++語言在不同的 SKILL 程序介面相關聯。

語言詳細說明

詳細說明語言的來源碼

來源碼檔案，檔案擴充的說明這語言



在交互式會議中語言詳細的說明

用不參與來接受 SKILL 函式。相互做用地，你可以請求 SKILL 或 SKILL++其中一個高水準的層次。

Language	Command
SKILL	toplevel 'il
SKILL++	toplevel 'ils

詳細說明語言在程式控制之下

對於高階程式人員，*eval* 函數在 SKILL 語法或 SKILL++語法之間其中一個計算符號。

變數範圍的比較

變數隨著記憶位置而的識別符號的關聯。參考環境是識別符的集合和他們關聯的記憶位置。變數的範圍提供你的程式的部分在提供變數相同位置裡。
在你程式執行之時依照某些範圍規則參考環境變數。甚至 SKILL 和 SKILL++兩者語言句法是完全一樣的。

- 使用不同的範圍規則
- 分割程式的不同

SKILL++使用詞彙語彙的範圍(Lexical Scoping)

詞彙範圍規則是只有關心你程式的來源碼。你程式的句子部分意思是在SKILL++函式間測試關聯的堵塞，像是 *let, letrec, letseq,*，和 *lambda* 函式，你可以在來源碼填入字塊。

SKILL 使用動態的範圍(Dynamic Scoping)

動態的範圍規則是只有關心你程式控制的平順。在 SKILL，你程式的句子部分意思是在計算符號的一段時間。通常，動態的範圍和詞彙的範圍規則一致。在這些事項裡，完全相同的符號在 SKILL 和 SKILL++裡都會回到相同值。

範例 1:有時範圍規則一致

以下規則增加行編號僅供參考

```
1: let( ( x )
2: x = 3
3: x
4: )
```

在 SKILL 和 SKILL++，*let* 函式爲 x 建立範圍並且這一式會回到 3。

- 在 SKILL++，x 的範圍是這字塊包括了第 2 列和第 3 列。在第 2 行
- 的 X 和在第 3 行的 X 是參考相同的記憶體範圍。

在 SKILL 中，x 的範圍開始當控制的平順輸入 *let* 函式和結束當控制平順 *let* 函式存在。

範例 2；當動態的詞彙範圍不一致

在範例 1.兩個範圍規則一致而且在 SKILL 和 SKILL++之間的 *let* 函式都回到相同的數值。範例 2 說明動態的和詞彙的不一致例子。注意到以下僅嵌入一項函數稱作是延伸先前範例在兩個參考 x 之間的加入一個 A 函數。然而，A 函數分配一個數值到 x。

```
1: procedure( A() x = 5 )
2: let( ( x )
3: x = 3
4: A()
5: x
6: )
```

考慮到 x 在第 1 行

- 在 SKILL++，x 在第 1 行和 x 在第 5 行參考了不同的區域位置因爲 x 在第 1 行是被外部字塊 *let* 函式決定的。*let* 函式回到 3。
- 在 SKILL，因爲第 1 行在執行 *let* 函式之時執行的，x 在第 1 行和 x 在第 5 行提供相同位置。*let* 函式回到 5。

例 3:稱作記憶體位置連續功能

在 SKILL，動態的範圍支配記憶位置影響依賴於函數稱作連續。在低於一級的符碼，函數 B 更新總體的不同的 x。但當自函數 A 呼叫，函數 B 改變函數 A 的位置代替多變的 x。函數 B 逐漸地更新 x，在 Alet 式。

- 在 SKILL 函數 A 回到 6
- 在 SKILL++函數 A 回到 5

```
procedure( A()
let( ( x )
x = 5 ;;; set the value of A's local variable x
B()
x ;;; return the value of A's local variable x
) ;let
) ; procedure
```

```
procedure( B()
        let( ( y z )
          x = 6
          z
            )
        ) ; procedure
```

在 12 頁提到 Dynamic Scoping 關於使用動態範圍的指引。

爲何詞彙範圍比可多次使用的符碼還好

重要的是變數的範圍不是非故意地中斷當程式修改或是其他重複使用的程式。通常細微的 bugs 產生於改變或重複使用在其他變數的延伸的設定。

程式在變數的領域裡應該可以視察來源碼而去決定功效。因爲動態的範圍緩和在執行你程式的過程，他可以自信地保護重複使用存在的符碼和因此動態的範圍衝擊可覆寫的組合碼。

摘要

SKILL++使用詞彙範圍。因爲詞彙範圍緩和僅僅在靜止的來源碼佈局。程式可以有信心地決定如何重複使用程式在變數位置的函式。

SKILL 使用動態領域。減輕 SKILL 程式可以有時非故意的中斷變數位置。介紹細微 bugs 的可能性則更高。

比較符號處理

SKILL 和 SKILL++共用同一個符號目錄表。每一個符號在兩種語言裡都是可以看見符號目錄表。

SKILL 如何使用符號

SKILL 有一種資料結構叫做符號。一個符號有一個不同且可辨認的名字和三個相關聯的記憶位置。在 91 頁有提到更多的資訊。

數值槽(The Value Slot)

SKILL 使用變數符號。變數一定會相同名稱的符號的數值槽。舉例來說，x＝5 儲存 5 值在符號 x 的數值槽。*Symeval* 函數回到數值槽的內容。舉例來說:
*symeval ('x)*回到 5 值。

函數槽(The Function Slot)

SKILL 使用符號函數槽去儲存函數物件。SKILL 計算函數叫做，就像:
fun(1 2 3)
吸引人的函數物件儲存在 *fun* 符號的函數槽。動態範圍一點也不影響所有函數槽。

屬性表(The Property List)

動態的範圍一點也不影響屬性表。關於重要的符號屬性表在 95 頁。

摘要

由於稱作 SKILL 函數 *set,symeval, getd, putd, get,* 和 *putprop,*你可以存取三個符號槽，以下的表格摘要 SKILL 操作影響三個符號的槽。

SKILL Construct	Value Slot	Function Slot	Property List
The assignment operator (=)	x		
set, symeval	x		
let and *prog* constructs	x		
procedure declaration		x	
getd, putd		x	
Function Call		x	
get, putprop			x

如何使用 SKILL++符號

正常來說，每一個 SKILL++全部的變數是限制跟符號函數槽一樣名字。這明顯隨著 SKILL 和 SKILL++共用函數。

有時候，你的 SKILL++程式需要存取 SKILL 全部的變數。靠著使用 *importSkillVar* 函式,你可以改變 SKILL++自符號函數槽到符號數值槽總體的裝訂。

在資料中函數使用的比較

在資料方面 SKILL++比 SKILL 處裡函數還簡單。

分配一個函數物件給一個變數

在 SKILL 中函數物件儲存在變數中而就像是其他資料數值。你可以使用相近於 SKILL 代數上的或者常見的語法函數去間接請求函數物件。例如，在 SKILL++中:

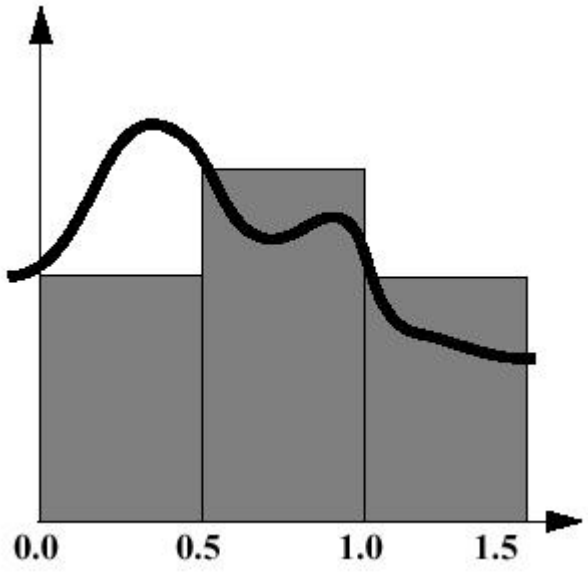
```
addFun = lambda( ( x y ) x+y ) => funobj:0x1e65c8
addFun( 5 6 ) => 11
```

在 SKILL 內，同樣的例子可以用兩種不同的方法完成，但是這兩種都比較不適宜。

```
addFun = lambda( ( x y ) x+y )
apply( addFun list( 5 6 )) => 11
or
putd( 'addFun lambda( ( x y ) x+y ))
addFun( 5 6 ) => 11
```

通過函數的要領

穿越函數在 SKILL++中不需要要求特殊的語法。在 SKILL 呼叫者必須引用函數名稱和被呼叫者必須使用 *apply* 函數去請求這通過函數。



AreaApproximation 函數在以下的例子估計在接近曲線的範圍之下，定義作 *fun* 函數超過距離(0 1)。這是接近由三個矩形的範圍增加入組成的，每一個都有寬 0.5 和高 fun(0.0), fun(0.5), and fun(1.0).

在 SKILL++中

```
procedure( areaApproximation( fun )
0.5*( fun( 0.0 ) + fun( 0.5 ) + fun( 1.0 ) )
) => areaApproximation
areaApproximation( sin ) => 0.6604483
areaApproximation( cos ) => 1.208942
```

在 SKILL 中

```
procedure( areaApproximation( fun )
0.5*(
apply( fun '( 0.0 )) +
apply( fun '( 0.5 )) +
apply( fun '( 1.0 ))
)
) => areaApproximation
```

areaApproximation('sin) => 0.6604483
areaApproximation('cos) => 1.208942

終止 SKILL++

SKILL++ *closure* 是一個函數物件包含一個或是更多 *free* 變數(定義在下面)限制資料，詞彙的範圍有可能終止。在 SKILL 中，動態的範圍有效率的預防終止使用。但 SKILL++應用能使用終止像軟體構建的封閉一樣。

空間(free)變數的關係

在來源碼片段之內，一個 *free* 變數是一個變數建立的你不可以決定測試來源碼。例如:考慮以下的來源碼衝突.

```
procedure( Sample( x y )
        x+y+z
        )
```

根據測試，x 和 y 不是空間變數的因爲他們是獨立變數。z 是空間變數的。在 SKILL++詞彙範圍意旨全部的函數的空間變數建立是決定在相同函數物件（關閉）是被建立時的。在 SKILL，動態的範圍意旨所有的參考自由變數是決定在執行時。

舉例來說，你能嵌入前文所定義的 *let* 函式連接 z 到 1。只有碼在相同的辭彙範圍內中 z 可以影響 z 的值碼。

```
let( (( z 1 ))
    procedure( Sample( x y )
        x+y+z
        )
    ;;; code that invokes Sample goes here.
  )
```

SKILL++的中止行爲是如何

SKILL++的應用能使用中止像軟體構件的程式塊。以下的例子增加複雜的說明來解釋 SKILL++如何中止。

範例 1：

```
let( (( z 1 ))
      procedure( Sample( x y )
            x+y+z
   )
      ;;; code that invokes Sample goes here.
   Sample( 1 2 )
   )
=> 4
```

範例 2：

```
let( (( z 1 ))
      procedure( Sample( x y )
            x+y+z
            )
      ;;; code that invokes Sample goes here.
   z = 100
   Sample( 1 2 )
   )
=> 103
```

這些例子分配 100 到 z 的建立。因此，這 *Sample* 函數回到 103.在這些範例，動態範圍和詞彙範圍一致。

範例 3

```
let( (( z 1 ))
      procedure( Sample( x y )
            x+y+z
            ) ; procedure
            ;;; code that invokes Sample goes here.
      let( (( z 100 ))
            Sample( 1 2 )
            ) ; let
      ) ; let
=> 4
```

這個例子中，從安置 *let* 函式之內請求 *Sample*。雖然動態範圍將會指示 z 限制到 100，而詞彙上它限制到 1。

範例 4

```
procedure( CallThisFunction( fun )
      let( (( z 100 ))
            fun( 1 2 )
        )
)
let( (( z 1 ))
      procedure( Sample( x y )
           x+y+z
   ) ; procedure
      CallThisFunction( Sample )
      )
=> 4
```

在這個 SKILL 範例裡，*let* 函式限制 z 到 1，建立 *Sample* 函數然後通過它去就到 *CallThisFunction* 函數。無論什麼時候 *Sample* 函數都是在執行，z 是限制到 1。特別是，當 *CallThisFunction* 請求 *Sample*，z 是限制到 1 甚至，既使 *CallThisFunction* 限制到 z，而在稱作 *Sample* 之前不一樣的值。

所以，Sample 爲了 *free* 變數的 z 已經壓縮這個的值。在 SKILL 做這個是不可能的，因爲動態範圍指示 *Sample* 將看見這個 Z 限制到到 100。

所以，SKILL++允許你建立一個你可以終止衝突的函數物件。某些行爲將會是最終稱作是獨立的一個種類。

範例 5

```
W = let( (( z 1 ))
lambda( ( x y )
x+y+z
)
)
=> funobj:0x1bc828
W( 1 2 ) => 4
```

在這範例裡，*Sample* 的名字是沒用的，因爲 *let* 函式它自己不包含呼叫 *Sample*。代替而是 *let* 函式回到函數物件。這函數物件是一個中止。這些碼回到不同的中止而每一個時間中這些碼被執行的。在 SKILL++，這裡是無法影響這些被限制 z。這些函數物件有效率地壓縮這些限制 z 的。

範例 6

makeAdder 函數在下方建立一個函數物件增加它是說明 x 加到變數 *delta*。每一個

稱爲 *makeAdde* 會回到不同的中止。

```
 procedure( makeAdder( delta )
        lambda( ( x ) x + delta )
        )
=> makeAdder
```

在 SKILL++，你能跳過 5 到 *makeAdde*r 和分配這結果到這變數的 *add5*。無論你多麼請求 *add5* 函數，他的變數 *delta* 位置也不會是限制到 5。

```
add5 = makeAdder( 5 )=> funobj:0x1e3628
add5( 3 ) => 8
let( ( ( delta 1 ) )
add5( 3 )
) => 8
let( ( ( delta 6 ) )
add5( 3 )
) => 8
```

SkILL++的環境

這一節介紹執行時資料架構稱作 *emvironment* 而 SKILL++使用上支援詞彙範圍環境。這節包含 SKILL++如何在執行時處裡環境。如果你要運用中止了解這性質是重要的。

爲求更多資訊，一起使用 SKILL 和 SKILL++放於 321 頁，包含如何檢視環境可以幫你爲 SKILL++的程式除錯。

執行中的環境

在計算你的 SKILL++程式的時候，建立變數的設定稱作環境。調和互相套疊的連續詞彙範圍，包含流通的 SKILL++狀態正計算著，環境是環境架構的表就像

- 在獨立的環境架構裡 SKILL++用相同的詞彙領域儲存所有變數。
- 相互套疊的連續詞彙彙領域相對於環境架構的名單。因此，每一個環境架構相同於兩個行列表。第一個行列包含多變的名字且第二個行列包含流動的值。

所以，每一個環境架構是等於兩個行列表。第一個行列包含變數的名字且第二個行列包含流動的值。

最高層次的環境

當 SKILL++交談式的分時系統開始，這動作的環境包含只有一個環境架構。那裡沒有其他的環境架構，所有的固定的函數和全面性變數的都在這個環境裡。這個環境就稱作最高層次環境。*toplevel('ils*)函數在初始化使用 SKILL++最高等級環境。然而它可能稱作 *toplevel('ils)*函數和通過非最高層次環境。考慮到函式就像

let(((x 3)) x) => 3

在此我們輸入一個叫做 *toplevel('ils)*。在互動之間我們企圖去檢索 x 值，然後調整它。參考 x 影響 SKILL++最高層次。

```
ILS-<2> let( (( x 3 )) toplevel( 'ils ) x )
ILS-<3> x
*Error* eval: unbound variable - x
ILS-<3> x = 5
5
ILS-<3> resume()
3
ILS-<2>
```

比較以下我們稱作 *toplevel* 通過在詞彙中止（動態的）環境。因此 *toplevel* 函式可以做存取非最高層次的環境。

```
ILS-<2> let( (( x 3 )) toplevel( 'ils theEnvironment() ) x )
ILS-<3> theEnvironment()->??
(((x 3)))
ILS-<3> x
3
ILS-<3> x = 5
5
ILS-<3> resume()
5
ILS-<2> x
*Error* eval: unbou nd variable - x
ILS-<2>
```

建立環境

在你的程式計算之間，當 SKILL++計算某些影響詞彙環境的函數時，SKILL++分配一個新環境架構並在增加它到動態環境之前。當這架構存在時，環境架構是從動態環境被分離的。

例題 1.

```
let(    ( (x 2)    (y 3) )
    x + y
    )
```

無論何時，SKILL++遇到已知語法，SKILL++會分配給一個環境訊框 (environment frame) 並且增加前面地操作環境 (active environment) 。

環境訊框

Variable	Value
x	2
y	3

求 x + y 的值，在前面地開始表列中，可以在 SKILL++環境訊架中查出 x 和 y 的值。當指令結束時，SKILL++會從操作環境上移除。只要有在使用到環境訊框時，將會在記憶體中繼續執行。

在這簡單的情況中，沒有人會這樣子使用，因此可能會遭到淘汰。

例題 2.

```
let(    ( ( x 2 )    ( y 3 )
    let(    ( ( u 4 )    (v 5 ) ( x 6 ) )
        u*v + x*y
        )
    )
```

在這次的 SKILL++中準備去求出 u*v + x*y 值，在前面地操作環境中有兩個訊框。

最外的環境訊框

Variable	Value
x	2
y	3

最內部的環境訊框

Variable	Value
u	4
v	5
x	6

經由環境訊框表列中可以明確地查出變數位置，注意在這兩個訊框中都有 x 變數值，在這個 u*v + x*y 的函數中，是選擇 x 值等於 6。

函數和環境

何時建立一個函數

SKILL++根據一個動作建立一個函數，連結到環境再分配給函數值。

何時呼叫一個函數

SKILL++建立一個環境函數變數值，再分配給一個新環境訊框。

```
procedure ( example ( u v )
    let (   ( ( x 2 )   ( y 3 ) )
        x*y + u*v
        )
    )
example ( 4 5 )
```

函數設定型式要和上面操作環境一樣，

環境訊框

Variable	Value
u	4
v	5

無論函數式是否是第一次被建立，都一定會被儲存起來的。

函數回傳值

SKILL++由操作環境移除環境訊框，並且在這例子中，這環境訊框變的無價值。在函數呼叫之前，就要交還到操作視窗上操作。

不變的環境

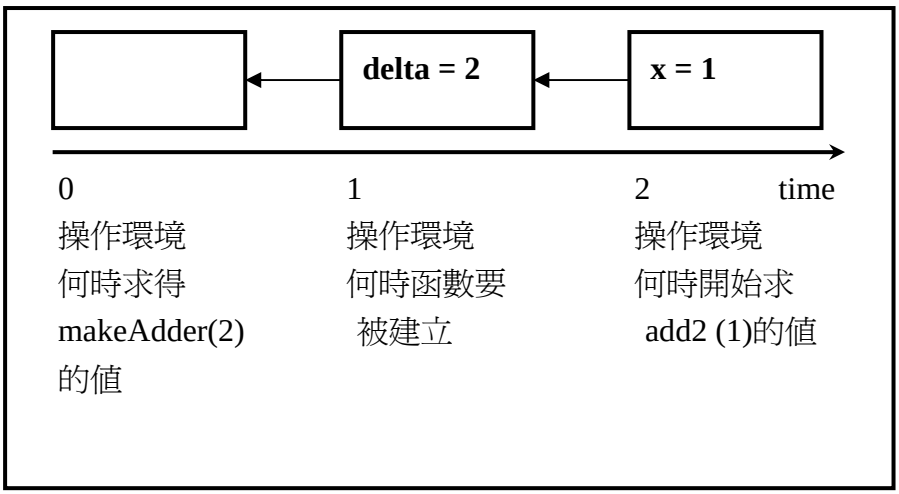
在這 makeAdder 例子裡秀出一個函數並且傳回一個值。在函數目的物被建立後才能傳回到這操作環境。傳回值並不影響目前的環境架構。

因此，無論何時你都可以使用回傳值，即使這函數已經有回傳值存在仍可以在接受回傳值。

SKILL++程式的優點就是使用過的程式都還可以被拿來使用，所以就能構成強大的軟體作業作程序。

```
procedure ( makeAdder( delta )
    lambda ( ( x ) x + delta )
    )
=>makeAdder
add2 = makeAdder ( 2 )   => funobj  ：0x1e6628
add2 ( 1 )   => 3
```

makeAdder 的傳回值動作圖解說明。



再呼叫一次 makeAdder 傳回到另一個函數。
add3 = makeAdder (3) => funobj：0x1e6638

在不同的環境下計算，以環形圍繞的環境是屬於 add2 和 add3 函數，灰色部份是操作環境的進入點，其它的環境是屬於 add3 函數的，假如 add3 被呼叫只會動作一次。