4

资料结构

1.存取运算符

有几个资料存取（access）运算符有类推（generic）的特性，换言之，同样的语法，可以用于不同资料型态资料的存取。以下是三个运算符：

■ -> 运算符

此一运算符可用于 disembodied property lists，defstruct，association table，和用户自定义型态。用以去参考某个 property 的值。

■ ~> 运算符

此运算符是前述箭头运算符的更广义的定义。 当直接用在一个件时，其作用与箭头运算符相同，但此一运算符也可以接受串列。

■ [ ] 运算符

此运算符为阵列存取文法运算符，可用来存取阵列的元赤素，或是关联式串列的“键一值 " 对的内容。

2.符号

在 SKILL 里面符号（symbol）与变数是同样的意思，经常交替使用。每个符号包括四个内涵（或是存放位置）： 显示名称（print name）、值（value）、函式链结（function binding）、属性串列（property list）。 除了名称之外，其他项目都是可有可无，而且通常不建议同时给一个符号值和函数链结。

2.1 产生符号

当在程序中第一次写到一个符号时，系统便自动建立此一符号。 当系统建立一个新的符号时，其值是设为 *unbound*。 所一般情况使用者不用做明显的建立符号的动作，不过 SKILL 仍提了一些符号建立的相关函数。

例如，使用者可以用下列函数产生新的符号

gensym（'a） Ð a1

gensym（'a） Ð a2

给定一个基本名称 *gensym* 函会产生一新的符号，同时为了避免重复，系统会自动判断在基本名称之后加上索引数值，来当做新符号的全名。 所产生的新符号对应的值是 *unbound*。

另外，也可以用 *concat* 函数来产生新的符号，如果用户想連合几字串来 产生新的符号名称的话。

2.2 符号的显示名称

符号名称可以包括文数字（*a~z*，*A~Z*，0~9），底线（__），问号（？）。如果名称的第一字符是数字，则其前须放置一个倒斜线（＼）字符。 事实上，非前述的字符也可用在符号名称中，但是每一个字符的前面要加上一个倒斜线。

吾人可以使用 *get_pname* 函数来取得符号的名称。此一函数看似多余，但当程序的处理中用到一个变数的值是“符号 " 时，就很管用了。例如：

A='S111

get_pname（A） Ð A

2.3 符号之值

符号的值可以是任何型态的资料，包括“符号 " 型态。吾人可以使用＝运算

子来指定一个值给一个符号。 函数 *setq* 即相当于＝运算符，因此下面两式是 相同的

A = 200

setq（A 200） 而要取得一个符

号的值只要使用其名便可：

A Ð 200

如果要参考整个符号变数本身（不是只有值而已），则可以使用单引号运算符：

position = ‘A Ð A

也可以用间接的方式来指定一个值给一个符号，或是取得一个符号的值：

set（position 200）

symeval（position） Ð 200

SKILL 处理全域变数、区域变数和函数参数的方式和 C 语言不同，SKILL 的符号可以代表全域和区域变数，一个符号在任何时候都可以被使用，差别只是取得什么值。SKILL 视每个符号的值放在一个堆栈中，而符号的目前值就是堆迭的最上面的项目，改变目前的值就是改变堆栈最上面元素的内容。 而当程序控制流程改变进入了 *let* 或是 *prog* 表示式的执行时，系统便会塞入一个暂存值进入该符号的值所存放的堆栈中。

3.符号之函数链结

当吾人宣告一个 SKILL 函数时，系统便用此函数的名称来建立一个符号，并用以存放函数的定义。 而当我们重新定义一个函数时，同样的名称符号不变，只是之前存放的函数内容定义会被弃掉。

与符号的值不同的是，函数定义的资料不会因为进入或离开 *let* 或 *prog* 表示式的执行而有所改变。

4.符号的属性列

属性列包括了所有的“属性项目一值 " 的配对。每一个“名称一值 " 对以相邻的两元素的方式存放在属性列里面，属性名称必须是符号名称，属性的值则可能是任意资料型态。

4.1 基本存取函数

每当一个符号产生时，SKILL 便会自动为其配上一个属性列，其起始内容设为 *nil*。 欲设定一个符号的属性列可以用下列函数

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

要取得一个属性列的内容可用下列函数

plist（'A） Ð （x 100 y 200）

要取得一个符号的某一个性质，可以使用点运算符

A.x Ð 100

getqq(A X) Ð 100

getqq(A Z) Ð nil 点运算符不可以巢狀方式使用，在点运算符的左右两侧必定是符号。 点运算符 对应的函数是 *getqq*。 如果你试图去取得一个不存在的性质的值，则所得到的 回传值会是 *nil*。 而如果你指定一个值给一个不存在的属性时，则该性质会被加 入属性列之内。

使用箭头 (->) 运算符和指定 (=) 运算符，可提供一个简单方式来进行非 直接地存取性质到一个符号的属性列的动作。如下例：

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

putpropq 函数的作用相当于箭头运算符加上指定运算符，上例的最后两叙述的 作用相同。

4.2 重要考虑

基本上，每一个符号对应的性质列都是全域性的， 无论何时当你传递一个符号给一个函数，该函数便可以異动符号的性质列的内容。 考虑以下的例子：

```
A = 1
let（A）
    A = 0
    A.x = 5
    ）；  let
x          Ð  1
plist（’x）Ð（example 5）
```

当程序执行离开了 let 算式之后， x 的值回復为原来的值，但是属性列的内容仍然保留改变之后的结果。

5.不具本体之属性串列

一个无本体之属性列（disembodied property list）在逻辑上可看成类似 C 语言里面的结构（structure），但不同于 C 的是，在不具本体之属性列中，我们可以动态地增加或移除位域。而箭头运算符也可以用来存取串列中的项目。

在底下的例子中，一个无本体属性列被用来表示复数。

```
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）  Ð（nil imag 7 real 3）
complex2 = createComplex（?real 2 ?imag 9）    Ð（nil imag 9 real 2）
i = createComplex（?imag 1）                   Ð（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）       Ð （nil   imag   0   real   -1）
```

在某情况之下使用无本体属性列有其好处。 例如有时候你想建立一个属性列但没有适当的符号来做为依附的本体。 有时候如果你为每个记錄动作都建立一个符号，有时可能会耗掉大量的符号，SKILL 在管理变数上会更没有效率。另一个原因是传递无本体属性串列的参数较传递符号容易。

值得一提的是，你可以将无本体属性串列当成值指定给一个符号，但这不会影响此一符号原有的属性列。

5.1 重点考量

在指定一个无本体属性串列给一个变数时， 要特别注意一件事。我们举下例说明：

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

由此可見 comp1 与 comp2 的内容完全相同，事实上，SKILL 是以指标的方式将二者都指向同一内存地址，如果用箭头运算符去修改 comp1 的 real 属性项 目，则 comp2 的 real 属性项目也会同步改变。要避免此种情况，须用 copy 函 式：

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

5.2 其他属性串列函数

要增加一个新的性质到一个符号的属性列，或是一个无本体的属性串列之中，可以使用 *putprop* 函数。 如果欲加入的性质项目早已存在，则 *putprop* 函

式会将该项目的旧值换成新值。 *putprop* 函数是一种 *lambda* 函数，换言之，呼叫此一函数时所有的实际引数项目都会先计算完后再代入形式引数。 而如果要取得一个有命名的属性串列的某一属性的值的话，要使用 *get* 函数。 *get* 的作用与 *putprop* 刚好相反。

举例如下：

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

第一跟第二行叙述的作用是相同的。 要增加一些属性给一个符号或是无本体属性串列， 而又不想将函数的参数
先行计算出来的话， 可以呼叫 *defprop* 函数。 举例如下：

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

要除去属性列中的一个属性可用 remprop 函数，举例如下：

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

6.字串

字串可视为一维的字阵列，在本节中介绍 SKILL 提供常用之字串处理的函式。

6.1 字串串接

用 *buildString* 函数可以由一串字串来结合成一个大字串，在大字串用指定的分隔字符来連接原来的各个字串。 如下例：

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

由上面可知道 buildString 的最后一个引数是指定的分隔字符。 要串接几个字串来产生一个新的字串可以使用 strcat 函数，原有的字串不会受到影响：

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

要取第二个字串中的前 n 个字符，再附加到第一个字串之后来产生一个新的字串，可以使用 *strncat* 函数。原有的字串不会受到影响：

strncat（”abc” “defgh” 4） Ð “abcdefg”

6.2 字串比较

在此介绍三个字串比较的函数。 首先， 如果要依 “文字顺序 " 比较两个字串或者是符号的大小，可以使用 *alphalessp* 函数。 如果第一个引数比第二个引数小，则回传值是 t。而 *alphalessp* 常常搭配 *sort* 函数使用来排序一列字串。使用如下：

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

单纯要比较两个字串的大小可以用 strcmp 函数。 举例如下：

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

另一个函数是 alphaNumCmp，可提供比较两个字串或符号的大小，比较的方式可以是依 “文字顺序 " 或是 “数值大小 " 。如果第三个引数是 non-nil 的，而且前两个引数是表示纯数值的字串，则进行数值大小之比较。举例如下：

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

如果只要比较两个字串的前面 *n* 个字符，则可以用 *strncmp* 函数，其比较 结果可以看回传值，表示的方式与 *strcmp* 相同：

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

6.3 处理字串中之字符

如果要得知一个字串的长度，可以使用 *strlen* 函数：

strlen（”xyz”） Ð 3
strlen（”\001”） Ð 1

如果要取得字串中的第几个字符，可以使用 *getchar* 函数：

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

注意，getchar 回传的是取字符为名的“符号 " ，不是“字串 " 。

要取得字串 *str1* 中第一次出现字串 *str2* 开始的部份，可以使用 *index* 函 式。如果是要取得 *str2* 最后一次出现到最后的部份，可以使用 *rindex* 函数。

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

要取得一个字符在符号或是字串中的位置，可以使用 nindex 函数。

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

6.4 建立子字串

要拷具一个字串的一部份成为一个新的字串要用 *substring* 函数。如下例：

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

其中第二个引数代表开始取子字串的位置，第三个引数代表要取几个字符。

parseString 函数的作用是根据指定的分割字符， 将一个字串分解成若干个子字串。用法如下：

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

第二个引数是若有指定，则 SKILL 会用此引数当作切割字符来分断原字串。若没有给第二个引数，则以空格符做为分断字符。

6.5 大小写转换

要将字串中的小写字换成大写字可使用 upperCase 函数；反之，则使用 lowerCase 函数：

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

7. Defstructs

Defstructs 是几个变数的集合，这些变数的型态可以都不一样，但在同样的一个结构（structure）的名称之下被处理。 这种资料结构类似于 C 语言中的 *struct*。以下是一个 *defstruct* 的例子：

defstruct（s_name s_slot1 s_slot2） Ð t

一旦一个结构被建立起来，其行为模式与无本体属性串列雷同，但储存方式较有效率且存取时较短。 结构可以动态地增加新的项目，不过那些动态产生的项目存取的效率和空间的使用率不若静态宣告者。

假设 *struct* 是一个结构的例子，则

struct->slot

的意义是传回 *slot* 这个位域对应的值；

struct->slot = newval

的意义是指定新的值给 *struct* 的 *slot* 位域；

struct->?

会传回 *struct* 的所有位域名称；

strcut->??

传回 *struct* 的属性串列。

7.1 其他 defstruct 函数

要测试一个 SKILL 的对象是否是某种结构的例子，可以使用 defstructp 函式。 如果回传值是 t ，表示答案是肯定的；否则回传值为 nil 。以下是一个例子：

defstructp（x structA） Ð t

defstructp（x “structA”） Ð t

第二个引数是结构名称，可以是符号，也可以是字串型式。 显示一个结构的内容可以用 *printstruct* 函数。即使此一结构中包含有一个位域也是结构型态，此一函数也能递归地印出所有的位域资料。 要拷贝一个结构的内容给另一个新的结构可以使用 *copy_<name>* 函数，此一函数是当 *defstruct* 函数在建立一个结构时产生的。 如果要递归地拷贝一个 结构的内容，则要使用 *copyDefstructDeep*。

8 阵列

在 SKILL 中使用阵列必须先行宣告，这跟使用变数不同。SKILL 的阵列有几个特点：

- 阵列中的元素可以分属不同型态
- SKILL 提供执行时期的阵列范围检查
- 阵列本身是一维的，但你可以建立阵列的阵列，如此相当于二维的阵列，以此类推，你可以建立更高维的阵列。
- 阵列范围检查是在执行时期，每一次有人使到阵列时就会进行的动作

8.1 配置指定大小空间给阵列

使用 declare 函数可以配置指定大小的空间给一个阵列。 如下例：

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

上述的 type 函数的作用是传回变数的型态。 要注意阵列的元素编号是由 0 开 始的。

当一个阵列的名称出现在一个指定叙述（＝号叙述）的左边， 而并未带有索引时，只有阵列对象本身被指定给别的变数， 阵列内所存的值并没有做拷贝的动作。 因此有可能给同一个阵列不同的名称。 这种情形有点像在 C 语言 中用一个指针 *b* 去等于一个阵列 *a* ，则不管用 *a* 或 *b* 都可以去存取同一个阵 列的内容。

下面是一些应用和说明：

```
declare（a[5]）      ；宣告含有五个元素的 a 阵列
b = a              ；b 指向 a 阵列 declare（c[4]）
                     ；宣告含有四个元素的 a 阵列
c[0] = b           ；c[0]指向 a 和 b 共同指的阵列位置
c[0][2]            ；  读取 a 阵列的第 3 个元素
```

9.关联式表格

关联式表格（association tables）是一堆“键 / 值 "（key/value）对的集合。可以被用做键者有整数、浮点数、字串、串列、符号，和些许用户定义的资料 型态。使用关联式的表格，其资料找寻的效率和方便性较使用无本体属性串列、阵列等要高。

9.1 初步化表格

使用 make T able 函数可以定义并初始化一个关联式表格。 这个函数接受的第一个引数是一个字串，做为列印用的表格名称，第二个引数则是可有可无的，如果有的话代表内定值，每当读取表格时所使用的键值不存在表格之中时，此一内定值便会被传回。考虑下例：

```
table1 = makeTable(”table1” 0)        Ð  table:table1
tablep(table1)                        Ð  t
table1[1] = “blue”                    Ð  “blue”
table1[“two”] = ‘( r e d )            Ð  (r e d)
table1[3]                             Ð  0
length(table1)                        Ð  2
```

其中 *tablep* 函数验证 table1 是否为一个表格； *length* 函数则是算表格内 有多少键值。

9.2 处理表格

SKILL 提供了一些处理关联式表格的函数。分别介绍如下： 首先，吾人可以使用 *tablep* 函数来检查某一个资料项是否为表格：

```
table1 = makeTable(“table1” 0)        Ð  table:table1
tablep(table1)                        Ð  t
tablep(5)                             Ð  nil
```

如果要将关联表格的内容转换成关联串列，则使用 *tableToList* 这个函数。此一转换对资料处理的效率而言，并不有利，因此最好别做这样的转换；相反地，只要利用它来逐步地查看表格的内容即可。

```
tableToList( table1)
    Ð  ((”two”（r e d）（1 “blue”))
    Ð
```

要将关联表格的内容写到档案去要用 *writeTable* 函数；要将一档案内容读入附加到一个关联表格之内，要用 *readTable* 函数。举例如下：

```
writeTable（”out1.log”  table1）      Ð  t
readTable（”out1.log”  table）        Ð  t
```

要将关联表格的内容印出成表格形式，则可使用 printstruct 函数。此一函式会递归地印出整个表格的内容。

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

9.3 浏览表格

用户可以使用 *foreach* 函数来查询表格中的每一个键值， 使用此一函数 的结果是印出表格中所有的 “键 / 值 " 对。另一个类似的函数是 *forall*，如果在表格中的每个键都是字串，值都是整数的话，便可以使用 *forall* 函数。

```
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* 函数的作用是，检查是否 *key* 是存在 *table*1 之内的键值，若有的话 就执行后面的叙述。

10. 关联串列

所谓关联式串列（association list）其实就是 “键 / 值 " 对的串列。一个关联串列是一个串列的串列。举例如下：

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

吾人可以用 *assoc* 函数来读取一个关联串列的项目：

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

assoc（”c” assocList） Ð （”c” 3）

吾人也可以用 rplaca 函数来更关联串列的项目： rplaca（cdr（assoc（”b” assocList））”two”） Ð （”b” “two”）
assocList Ð（（”a” 1）（”b” “two”）（”c” 3））