3

建立 SKILL 函数

1.基本概念

在第一意之中吾人已大致介绍函数的基本观念。 在本章中则更进一步告诉使用者如何去定义一个函数，以及全域或区域变数。

事实上，在 SKILL 里面提供了不同型式的函数，分别是 *lambda*，*nlambda*，及 *macro* 三类。SKILL 以不同的方式来处理这三种函数：

- 大部份我们所定义的函数属 lambda 函数。 SKILL 会先将参数的值赋与 函数的形式参数（formal parameter），然后再计算函数的值。
- 有一些 SKILL 的内部函数是所谓的 *nlambda* 函数，此种函数只有单一的形式引数（formal argument）。当呼叫一个 *nlambda* 型的函数时，SKILL 会将所有传入的真实参数集合成一个串列。然后指定给唯一的形式引数。
- macro 函数与 lambda 函数的文法雷同，但内涵则大不相同。 Macro 的计算是在编译时期进行的，而非执行时期。

2.文法函数

SKILL 提供一组文法函数，使用者可以利用这些函数来定义新的函数。大部份时候你应该使用其中的 procedure 或 defun 函数。

2.1 procedure

使用 procedure 来定义函数大概是最普遍的方式，任何用其他方式定义的函式都可以用 procedure 的函数来定义之。下面是一个例子：

```
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
    ）    Ð  trAdd
trAdd(6 7)        Ð  13
```

2.2 lambda

lambda 函数可以定义一个没有名称的函数。它的回传值就是一个函数的对象，可以被指定给一个变数来储存。例如：

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

而已宣告之函数对象可以传递给 apply 函数来执行：

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

通常吾人不太会用到 lambda 函数的宣告，只要用 procedure 函数即可。

2.3 其他文法函数

除了上述两个文法函数之外，尚有几个特殊之文法函数。 n*procedure* 函式是针对旧的程序版本要升级时，为了兼容性考虑才会用到的， 在新完成的程序中不应使用到。此一函数允许你的函数用 procedure 宣告加上@rest 选项的方式， 来接收暂订数目的引数。 也允许使用者用 *defmacro* 函数来接收未代入值之引数。

至于 *defmacro* 函数则提供了一个定义 *macro* 函数的方法。 你可以用 macro 来定义自己风格的 SKILL 语法，而 macro 则负责在编译时期将你自订的文法转成 SKILL 的普通叙述，以供后续之编译与执行。

mprocedure 则是 defmacro 另一个更基本的替代选择。 *mprocedure* 只有单一个引数，整个用户自己的语法型式是完整不动地传给 *mprocedure* 函数的。 不要用此一函数在新写的程序中，它主要的目的是供修改旧版的

程序中的函数，以与旧版的系统兼容。 在新写的程序中请用 *defmacro* 来定义函数，如果你必需去接收一些未定数目的、未代入值的引数的话，可以使用*@rest* 参数。

2.4 综合整理

下表是针对文法函数的内容作一整理：

文法函数	函数型态	引数计算（**evaluation**）	执行
Procedure	lambda	实际引数的值被计算出来，并且传给对应之形式引数（formal arguments）	在函数中的表示式是在执行时求出其值，并回传最后一个叙述的值
Defmacro	macro	实际引数的值没有被计算出来，直接将表示式传给对应之形式引数	在函数本体内的每个表示式在编译时期会先被展开，而最后的结果才会被编译
Mprocedure	macro	整个函数调用都被对应到单一个形式引数	在函数本体内的每个表示式在编译时期会先被展开，而最后的结果才会被编译
Nprocedure	nlambda	所有的实际引数都不会被展开计算，并被集合成一个串列，再对应给单一个形式引数	在函数中的表示式是在执行时求出其值，并回传最后一个叙述的值

3.定义参数

使用者可以透过在形式引数之中加入一些@选项来决定实际引数要如何被传 给形式引数。 @的选项主要有三个，*@reset*、*@optional*、*@key*。

3.1 @reset 选项

使用@reset 选项可让用户在呼叫函数时可以传递任意数目之参数（存在一个串列之中）给这个函数。 下面的例子显示使用@reset 的好处：

```
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）  Ð  6
```

结果在 CIW 上显示

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

传递给 trTrace 的参数个数在不同的呼叫中皆可不同。

3.2 @optional 选项

@optional 提供使用者另一种可指定不同参数个数的函数使用方法。 同时，使用者也可以对每一个参数指定一个内定值， 使得当呼叫函数未指定某一参数的值时，SKILL 可以指定内定值给此一参数； 如果没有预设参数的内定值，则其内定值自动设为 *nil*。

如果使者在 procedure 的引数列定义里面放了 *@optional* ，则任何在其后的参数都是 “可给值，可不给值 " 的。以下是范例程序：

```
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，当呼叫函数时引数 *h* 与 *w* 必须传值给它，但是 *x* 与 *y* 是选择性的 参数，可传值也可不传值给它。 如果没有传的话，则 *x* 或 *y* 便会用内定的值， 也就是 0。以下是一些执行的结果：

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

3.3 @key 选项

@optional 是依照函数的引数顺序来判断那些实际参数指定给那些形式引数。若使用 *@key* 的选择项则可以让用户依不同的顺序来决定那些引数要传 值给它，那些不要。以下是一个例子：

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

注意的是，*@optional* 与*@key* 是互斥的，它们不能同时出现在一引数列。

4.型态检查

不像一般传统的程序语言是在编译时期做型态检查的动作，SKILL 是在函式被执行时才做动态的型态检查。 每一个 SKILL 的 *lambda* 或 *macro* 函数都可以在宣告引数的串列中加入一个“引数样板 "（argument template），以定义所要求的引数资料型态。 但 *mprocedure* 函数并没有此种功能。

在引数样板使用的字符代表不同的资料型态，这在第二章中有提到，除基本的之外，使用者也以用代表“组合 " 型态的字符符号，来代表两种以上的型态。表列如下：

字符	代表意义
S	符号或字串
N	数值，包括定点及浮点数
U	函数－ 不管是函数的名称，或是 lambda 函数的本体（list）
G	任何资料型态

以下是例子：

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

此时 “nn”代表函数 *f* 接受两个数值的引数。

5.批注函数

SKILL 提供一种方法来宣告一个批注，当你在建立一函数的时候，做法如下例：

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

其中在引数宣告之后的第一行字串即为批注。

6.区域及全域变数

6.1 定义区域变数

SKILL 提供一个 *let* 函数来建立一暂存值给区域变数。 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* 是宣告的区域变数，其初值是 *nil* 。使用 let 函数可以启 始一个非 nil 值的区域变数。下面是一个例子：

```
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* 函数也可以宣告区域变数，其语法如下：

prog（（local variables）your SKILL statements）

6.2 测试全域变数

应用程序通常会启始化一或多个全域变数。当一个应用程序第一次执行时，其全域变数通常是 *unbound* 的，在此情况试图去使用全域变数传回值会造成错误。

吾人可以使用 *boundp* 函数来检查一个变数是否是 *unbound*。举例如下：

boundp（'Items）&& Items

当 *Items* 是 *unbound* 时，上式传回 *nil*；否则，传回 *Items* 的值。

6.3 重新定义既有之函数

当进行侦错的动作时，使用者常常需要重新定义一个函数的内容。一般 procedure 定义的建构方式允许使用者去重新定义既有之函数，但前提是函数不得是已启动“写保护 " 的狀态。要启动 writeProtect 开关，必须执行下列函数：

sstatus（writeProtect nil）

除了侦错的目的之外，拥有对同样函数内容进行多复位义的能力，有时也是很方便的。例如，在开放式仿真系统（Open Simulation System）“内定 " 的网络（netlist）函数可以被使用者自定义的函数取代，便是一例。