```
Dyalog APL/W Version 9.0
Function Results Edition
Tue Oct 17 16:54:50 2006
clear ws

      ⍝ In what follows, note the difference between
      ⍝ <naming a function> and <naming its result>:

      dup ← {⍵ ⍵}          ⍝ name a D-function

      vis ← {⍵ ⍵} 20       ⍝ name the result of a D-function

      dup                  ⍝ show the D-function
{⍵ ⍵}
      vis                  ⍝ show the result
20 20
      ⍝ In this experimental version of the interpreter,
      ⍝ a dyadic function with a missing right argument
      ⍝ binds its left argument to form a monadic function.
      ⍝ This phenomenon is known as "currying":

      doub ← 2 ×           ⍝ name a doubling function

      doub 5               ⍝ apply the function
10
      ⍝ Secondly, a D-function can return a FUNCTION,
      ⍝ so we can parameterise the above definition.

      ⍝ To avoid confusion, we will distinguish function-returning functions
      ⍝ with names ending in ∆. The ∆ has no significance for the interpreter.

      mul∆ ← {⍵×}          ⍝ name a function-returning function

      trip ← mul∆ 3        ⍝ name a tripling function result
      half ← mul∆ ÷2       ⍝ name a halving function result

      half trip doub 5     ⍝ apply functions
15
      (mul∆ 6) 7           ⍝ return and apply function
42
      doub                 ⍝ show function                         ---.
2 ×
      trip                 ⍝ show function                         ---'
3 ×
      half                 ⍝ show function
0.5 ×
      mul∆                 ⍝ show the function-returning function
{⍵×}
      ⍝ <uns> is a regular D-function, often used with ⎕nread,
      ⍝ to "cast" an ⍺-bit integer array to unsigned numbers:

      uns ← {base←2*⍺ ⋄ base|base+⍵}      ⍝ unsigned numbers      ---.

      ⎕ ← nums ← ¯3+⍳5                    ⍝ test sequence            ·
¯2 ¯1 0 1 2

      8 uns nums                          ⍝ cast to  8-bit unsigned  ·
254 255 0 1 2

      16 uns nums                         ⍝ cast to 16-bit unsigned  ·
65534 65535 0 1 2
```

```
      ⍝ We can bind a special function for any particular word width:·

      uns8 ← 8 uns                    ⍝ bind an 8-bit cast       ·    ---.

      uns8 nums                       ⍝ cast to 8-bit unsigned   ·       ·
254 255 0 1 2

      ⍝ Alternatively, we can code an equivalent function-returning function:·

      uns∆ ← {base←2*⍵ ⋄ {base|base+⍵}}   ⍝ unsigned numbers     ---.       ·

      uns8 ← uns∆ 8                   ⍝ name an 8-bit cast       ·    ---'

      uns8 nums                       ⍝ cast to 8-bit unsigned   ·
254 255 0 1 2

      uns8                            ⍝ show returned function---|
 {base|base+⍵} ; base←256

      ⍝ <uns8> is a new type of function, called a "closure",        ·
      ⍝ · which is a function                                        ·
      ⍝ · · bound with the local environment                         ·
      ⍝ · · · of the D-function that returned it.                 ---'
      ⍝
      ⍝   Function ++ Environment => Closure

      ⍝ Here is another example: the inverse of <uns∆> is a function <int∆>,
      ⍝ which returns a function, which casts to signed integers:

      int∆ ← {base half←2*⍵-0 1 ⋄ {(base|half+⍵)-half}}          ⍝       ---.

      int8 ← int∆ 8                ⍝ name an 8-bit signed cast              ·

      int8 0 127 128 255           ⍝ cast to 8-bit signed                   ·
0 127 ¯128 ¯1

      int8                         ⍝ show the closure           ---.       ·
 {(base|half+⍵)-half} ; half←128, base←256

      ⍝ We say: The function {···}, "where" half is 128 "and" base is 256.   ·

      ⍝ Here's how it works:                                         ·       ·

      ⍝ When a D-function returns a function,                        ·       ·
      ⍝ · it binds a copy of its local environment to the result.    ·    ---'

      ⍝ When a closure is evaluated,                                 ·
      ⍝ · its bound environment is pushed onto the stack,            ·
      ⍝ · · installing local values and                              ·
      ⍝ · · · shadowing any items of the same name.                ---'

      ⍝ Some more examples ...

      bkt∆ ← {L←⍺ ⋄ R←⍵ ⋄ {L,⍵,R}}        ⍝ bracketing function   ---.    (Dfn)

      brack ← '[' bkt∆ ']'            ⍝ square-bracketed         ·    (Csr)

      brack'hello'                    ⍝ apply closure            ·
[hello]

      brack                           ⍝ show closure             ·
 {L,⍵,R} ; L←'[', R←']'
```

```
      paren ← '(' bkt∆ ')'                ⍝ round-bracketed           ·
      angle ← '<' bkt∆ '>'                ⍝ angle-bracketed           ·

      angle paren brack 'hello'           ⍝ apply closures            ·
<([hello])>

      ⍝ We could recode the bracketing function without local variables:

      bkt∆ ← { (⍺∘,)∘(,∘⍵) }              ⍝ bracketing function   ---'

      arrow ← '→' bkt∆ '←'                ⍝ arrow-bracketed

      arrow 'world'                       ⍝ apply function
→world←
      arrow                               ⍝ show function
 →∘, ∘ ,∘←

      ⍝ As there are no local names to bind, <arrow> is just a regular
      ⍝ derived function.

      ⍝ A closure is a <generalisation> of a regular function,
      ⍝ which means that a regular function is just a special case,
      ⍝ of a closure with no local names:

      iota ← ⍳                ⍝ regular (primitive) function      ---.

      iota ← {⍳} 123          ⍝ same as above                     ---'

      iota                    ⍝ (null environment)                ---.
⍳
      ⍝ Some more examples ...                                    ·

      di0 ← {⎕io←0 ⋄ ⍳⍵}      ⍝ regular D-function for origin-0 (monadic) iota

      ci0 ← {⎕io←⍵ ⋄ ⍳}0      ⍝ closure for origin-0 (ambi-valent) iota   ---.

      di0 4                   ⍝ apply the D-function              ·       ·
0 1 2 3
      ci0 4                   ⍝ apply the closure                 ·       ·
0 1 2 3
      di0                     ⍝ show the D-function               ·       ·
{⎕io←0 ⋄ ⍳⍵}
      ci0                     ⍝ show the closure                  ---'    ·
⍳; ⎕io←0

      )copy dfns cmpx
C:\dfns\dfns saved Thu Sep 28 17:12:46 2006

      cmpx'di0 4' 'ci0 4'     ⍝ compare evaluation times                  ·
  di0 4 1.6E¯6   0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  ci0 4 6.6E¯7 -59% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕

      ⍝ A closure may be viewed as a "partial evaluation".                ---'

      ⍝ Here is a handy operator:

      ml∆ ← {⎕ml←⍵ ⋄ ⍺⍺}      ⍝ returns ⍺⍺ where ⎕ml←⍵

      list ← ∊ ml∆ 1          ⍝ define enlist function

      hello←'hello' 'world'   ⍝ nested vector
```

```
      list hello              ⍝ apply enlist
helloworld

      list                    ⍝ show enlist
∊; ⎕ml←1

      cmix ← ↑ ml∆ 0          ⍝ define mix closure

      cmix hello              ⍝ apply mix closure
hello
world
      cmix                    ⍝ show mix closure
↑; ⎕ml←0

      dmix ← {⎕ml←0 ⋄ ↑⍵}     ⍝ define regular mix D-function

      ⎕vr ⎕fx 'z←tmix a;⎕ml' '⎕ml←0 ⋄ z←↑a'   ⍝ define a mix trad-function
     ∇z←tmix a;⎕ml
[1]   ⎕ml←0 ⋄ z←↑a
     ∇

      cmpx'tmix ⍬ ⍬' 'dmix ⍬ ⍬' 'cmix ⍬ ⍬' '   ↑ ⍬ ⍬'     ⍝ compare times
  tmix ⍬ ⍬ 3.0E¯6   0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  dmix ⍬ ⍬ 2.4E¯6 -20% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  cmix ⍬ ⍬ 1.4E¯6 -56% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
     ↑ ⍬ ⍬ 8.2E¯7 -73% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕

      ⍝ Here is a D-function, which returns
      ⍝ · the next ⍵ numbers in sequence,
      ⍝ · · by updating a <global> variable:

      N←0                                      ⍝ initialise global variable

      s ← {(⍳N+←⍵)+N}                          ⍝ next ⍵ numbers in sequence.

      s 2                                      ⍝ next 2 numbers.
1 2
      s 3                                      ⍝ 3 more numbers.
3 4 5
      s 2                                      ⍝ 2 more numbers
6 7
      N                                        ⍝ value of global variable
7
      ⍝ We can make N local to a closure:

      ⎕ex'N'                                   ⍝ remove global variable

      s∆ ← {N←⍵ ⋄ {(⍳N+←⍵)+N}}                 ⍝ sequence function

      s ← s∆ 0                                 ⍝ sequence closure

      s                                        ⍝ show the closure         ---.
 {(⍳N+←⍵)+N} ; N←0

      s 3                                      ⍝ next 3 numbers              ·
1 2 3
      s 2                                      ⍝ 2 more numbers              ·
4 5
      s                                        ⍝ show the closure         ---|
 {(⍳N+←⍵)+N} ; N←5

      s 2                                      ⍝ 2 more numbers              ·
6 7
```

```
      s                                       ⍝ show the closure         ---'
 {(⍳N+←⍵)+N} ; N←7

      ⎕nc'N'                                  ⍝ look: no global N
0
      ⍝ We say: N is a "Persistent Local Variable" of function s.

      ⍝ Here is another example of the use of a persistent local variable:

      fib ← {⍵∊0 1:⍵ ⋄ +/fib¨⍵-1 2}   ⍝ naïve coding for ⍵th Fibonacci number

      fib¨ time ⍳30                   ⍝ ··· is rather slow                :-(
13.50
      )copy demo memoΔ
C:\demos\demo saved Sun Oct 22 12:23:30 2006

      memoΔ
{                             ⍝ Memoization operator.
    to fm←⍺ ⍵                 ⍝ result values and keys.
    ⍺⍺{                       ⍝ ⍺⍺ is subject function.
        (⊂⍵)∊fm:(fm⍳⊂⍵)⊃to    ⍝ arg known: corresponding rslt.
        ⊃θρ⌽fm to,∘⊂←⍺⍺\⍵ ⍵   ⍝ otherwise: calculate & remember.
    }
}
      fib ← θ fib memoΔ θ             ⍝ "memoized" Fibonacci

      fib¨ time ⍳100                  ⍝ ··· is somewhat quicker
00.01
      ⍝ Further, we can let memoΔ take care of the base cases of the recursion:

      fib ← {+/fib¨⍵-1 2}             ⍝ naïve coding for ⍵th Fibonacci number

      fib ← 0 1 fib memoΔ 0 1         ⍝ "memoized" Fibonacci

      fib¨ time ⍳1000                 ⍝ also quick
00.17
      ⍝ Closures may be nested ...

      sum ← {a←⍵ ⋄ {b←⍵ ⋄ {c←⍵ ⋄ {a+b+c+⍵}}}} ⍝ dfn → csr → csr → csr → array

      sum                                     ⍝ sum is a D-function,
{a←⍵ ⋄ {b←⍵ ⋄ {c←⍵ ⋄ {a+b+c+⍵}}}}

      sum 1                                   ⍝ · which returns a closure,
 {b←⍵ ⋄ {c←⍵ ⋄ {a+b+c+⍵}}} ; a←1

      (sum 1)2                                ⍝ · · which returns a closure,
 {c←⍵ ⋄ {a+b+c+⍵}} ; b←2; a←1

      ((sum 1)2)3                             ⍝ · · · which returns a closure,
 {a+b+c+⍵} ; c←3; b←2; a←1

      (((sum 1)2)3)4                          ⍝ · · · · which returns an array.
10
      ⍝ Let's extend our sequence function,
      ⍝ · to return a closure,
      ⍝ · · which returns a <pair> of sequences:

      ssΔ ← {N n←⍵ ⋄ {(⍳¨N n+←⍵)+N n}}        ⍝ sequences function

      ss ← ssΔ 100 0                          ⍝ sequences
```

```
      ss 3 2                                  ⍝ next 3 and 2 numbers
101 102 103  1 2

      ss 3 2                                  ⍝ 3 and 2 more numbers
104 105 106  3 4

      ss                                      ⍝ show closure
{(⍳¨N n+←⍵)+N n} ; n←4, N←106

      ⍝ Next, let's localise variables N and n in separate closures:

      ssΔΔ ← {N←⍵ ⋄ {n←⍵ ⋄ {(⍳¨N n+←⍵)+N n}}} ⍝ sequences function function

      ssΔ ← ssΔΔ 0                            ⍝ sequences function

      ssΔ                                     ⍝ show closure           ---.
{n←⍵ ⋄ {(⍳¨N n+←⍵)+N n}} ; N←0
                                                                          ·
      ⍝ Remember how closures work:                                       ·

      ⍝ [1] When a D-function returns a function,                         ·
      ⍝     · it binds a copy of its local environment to the result.     ·

      ⍝ [2] When a closure is evaluated,                                  ·
      ⍝     · its bound environment is pushed onto the stack,             ·
      ⍝     · · installing local values and                               ·
      ⍝     · · · shadowing any items of the same name.                   ·

      ⍝ [2] => ssΔ's bound variable N is local while ssΔ is evaluated     ·
      ⍝ [1] => and so N is bound to any function returned by ssΔ.         ·
      ⍝ Ergo:  closures returned by ssΔ <share> ssΔ's bound N.         ---'

      ⍝ New variables are created only as D evaluates a ←.

      ssΔ                             ⍝ show ssΔ again          ---.
{n←⍵ ⋄ {(⍳¨N n+←⍵)+N n}} ; N←0

      s1 ← ssΔ 0                      ⍝ sequences                  ·

      s1                              ⍝ show closure            ---|
{(⍳¨N n+←⍵)+N n} ; n←0; N←0

      s1 3 2                          ⍝ next 3 and 2 numbers       ·
1 2 3  1 2

      s1                              ⍝ show closure               ·
{(⍳¨N n+←⍵)+N n} ; n←2; N←3

      ssΔ                             ⍝ s1 shares ssΔ's N       ---'
{n←⍵ ⋄ {(⍳¨N n+←⍵)+N n}} ; N←3

      s2 ← ssΔ 0                      ⍝ make another sequences

      s2                              ⍝ show closure
{(⍳¨N n+←⍵)+N n} ; n←0; N←3

      s2 2 3                          ⍝ next 2 and 3 numbers
4 5  1 2 3

      ssΔ                             ⍝ s2 also shares ssΔ's N
{n←⍵ ⋄ {(⍳¨N n+←⍵)+N n}} ; N←5
```

```
      s1 ⋄ s2                         ⍝ show both closures
 {(⍳¨N n+←⍵)+N n} ; n←2; N←5
 {(⍳¨N n+←⍵)+N n} ; n←3; N←5

      ⍝ Closures accumulate incrementally as inner D-functions exit,
      ⍝ which means that inner function-returning functions incur
      ⍝ a minimum of runtime overhead. Here is a code snippet:
      ⍝   {
      ⍝       ...
      ⍝       fid ← 'c:\tmp\',⍵              ⍝ local value
      ⍝       ...
      ⍝       tie←{⍵:⎕fstie ⋄ ⎕ftie}share     ⍝ no inner locals => no closure.
      ⍝       ...
      ⍝       tno←fid tie 0
      ⍝       ...
      ⍝   }
      ⍝ We can display intermediate closures using ⎕←

      ⎕←{a←⍵ ⋄ ⎕←{b←⍵ ⋄ ⎕←{c←⍵ ⋄ ⎕←{a+b+c+⍵} }3 }2 }1         ⍝ show closures
{a+b+c+⍵}
 {a+b+c+⍵} ; c←3
  {a+b+c+⍵} ; c←3  ; b←2
   {a+b+c+⍵} ; c←3  ; b←2  ; a←1

      ⍝ Here is a more concrete example of the use of nested closures ...

      )copy demo Q∆∆
C:\demos\demo saved Sun Oct 22 12:23:30 2006

      Q∆∆                             ⍝ show D-function
{                       ⍝ Queues.

    Queue←⍵             ⍝ common buffer.

    {                   ⍝ outer closure.
        'put'≡⍵:{       ⍝ inner closure.
            Queue,←⊂⍵   ⍝ append value.
        }

        'get'≡⍵:{       ⍝ inner closure.
            nxt←⍵↑Queue ⍝ next ⍵ items.
            Queue↓⍨←⍵   ⍝ drop ⍵ items.
            nxt         ⍝ return items.
        }
    }
}
      Q∆ ← Q∆∆'one' 'two' 'three'     ⍝ <trace> queue primed with some vals

      ⎕nc'Queue'                      ⍝ look: no global Queue variable
0
      Q∆                              ⍝ show closure
 {··} ; Queue

      get ← Q∆'get'                   ⍝ <trace> generation of getter

      put ← Q∆'put'                   ⍝ <trace> generation of putter

      put¨ 'four' 'five' 'six'        ⍝ <trace> put some more values

      get 1                           ⍝ <trace> get first value
 one
      put'seven'                      ⍝ <trace> put another value
```

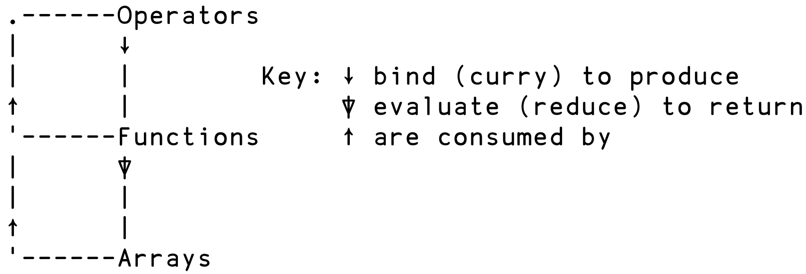
```
      get 3                         ⍝ <trace> get some values
 two  three  four

      KΔ ← Q∆Δ'en' 'to' 'tre'       ⍝ <trace> en anden kø med nogle numre

      tag ← KΔ'get'                 ⍝ genererer en "tager"

      giv ← KΔ'put'                 ⍝ genererer en giver

      tag 2                         ⍝ <trace> tager de næste numre
 en  to

      get 2                         ⍝ <trace> get some more values
 five  six

      giv'fire'                     ⍝ sætter et nummer ind

      put'eight'                    ⍝ put a value

      get 2                         ⍝ get last two values
 seven  eight

      tag 2                         ⍝ tager de sidste to numre
 tre  fire

      ⍝ It is often convenient to collect related functions into one capsule.
      ⍝ For example, each of the data-packing functions from dfns.dws contains
      ⍝ both pack (cmp) and unpack (exp) as sub-functions:

      )copy dfns packN
C:\dfns\dfns saved Thu Sep 28 17:12:46 2006

      packN                                         ⍝ ⍺ selects pack/unpack
{                          ⍝ Null packing.

    cmp←{
        mask←,⍵≢1↑0⍴⍵       ⍝ mask of non-nulls.
        (⍴⍵)mask(mask/,⍵)   ⍝ shape mask non-nulls.
    }

    exp←{
        shape mask items←⍵
        shape⍴mask\items
    }

    ⍺←1 ⋄ ⍺:cmp ⍵ ⋄ exp ⍵   ⍝ compress or expand.
}
      ⍝ We can get a handle on the inner functions by binding a left argument:

      cmp ← 1 packN                                 ⍝ compress function

      exp ← 0 packN                                 ⍝ expand function

      ⍝ Similarly, operator sbst derives a set of Simple
      ⍝ Binary Search Tree (BST) functions: get, put, rem, fmt, ...

      )copy demo sbst
C:\demos\demo saved Sun Oct 22 12:23:30 2006

      ⍝ sbst uses a left <operand> to distinguish the cases,
      ⍝ leaving ⍺ and ⍵ free for inner dyadic functions:
```

```
      sbst
{⎕io ⎕ml←0                                  ⍝ Simple Binary Search Trees.

    put←{                                   ⍝ tree ⍺ with key=value ⍵.
        ⍺≡0:(⍵(0 0))0                       ⍝ null: new node.
        ((nxt _)subs)(key _)←⍺ ⍵            ⍝ node and search key/val.
        nxt≡key:(⍵ subs)0                   ⍝ match: new value.
        ⍺ ∇ search ⍵                        ⍝ natch: try subtrees.
    }                                       ⍝ :: t _ ← t ∇ k v

    get←{                                   ⍝ value for key ⍵ from tree ⍺.
        ⍺≡0:⍺'?'                            ⍝ null: key not in tree: no value.
        ((nxt val)_)(key _)←⍺ ⍵             ⍝ node and search key.
        nxt≡key:⍺ val                       ⍝ match: tree & value.
        ⍺ ∇ search ⍵                        ⍝ natch: try subtrees.
    }                                       ⍝ :: _ v ← t ∇ k _

    rem←{                                   ⍝ tree ⍺ without key ⍵.
        ...
    rrot←{                                  ⍝ right rotation.
        ...
    search←{                                ⍝ search subtree ⍺ for key ⊃⍵.
        ...
    fmt←{                                   ⍝ formatted tree ⍵.
        ...
    vec←{                                   ⍝ vector of key=value pairs.
        ...
    chk←{                                   ⍝ tree stats / integrity check.
        ...
    bal←{                                   ⍝ dsw-balancing.
        ...
    cmp←{                                   ⍝ compress of alternate vine sections.
        ...
    list←{                                  ⍝ list (0-vine) from tree ⍵.
        ...

    'put'≡⍺⍺:⊃⍺ put ⍵                       ⍝ insert/replace value in tree.
    'get'≡⍺⍺:⊃⌽⍵ get ⍺ 0                    ⍝ search for value for key.
    'rem'≡⍺⍺:⊃⍺ rem ⍵ 0                     ⍝ remove key=value from tree.
    'fmt'≡⍺⍺:fmt ⍵                          ⍝ formatted tree.
    'vec'≡⍺⍺:vec ⍵                          ⍝ vector of key=value pairs.
    'chk'≡⍺⍺:0 chk ⍵                        ⍝ tree stats and integrity check.
    'bal'≡⍺⍺:bal ⍵                          ⍝ balanced tree ⍵.
}
      dput←'put'sbst                          ⍝ derive a put function

      tree←(0 dput 2)dput 4                   ⍝ new tree with nodes 2 and 4

      dfmt←'fmt'sbst                          ⍝ derive a fmt function

      dfmt tree                               ⍝ 2-node tree
2=2.
   '4=4

      dfmt((tree dput 1)dput 3)dput 5         ⍝ tree with three  more values
   .1=1
2=2|
   |   .3=3
   '4=4|
       '5=5
```

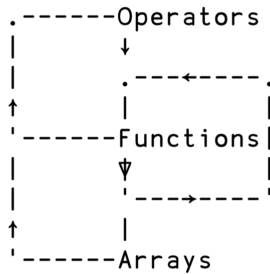
```
      dfmt tree dput 1                    ⍝ <trace> tree with extra node
   .1=1
2=2|
   '4=4

      )ed sbst                            ⍝ change to closure-returning cbst:

      )copy demo fndiff                   ⍝ show changes operator->closure
C:\demos\demo saved Sun Oct 22 12:23:30 2006

      fndiff'sbst' 'cbst'
sbst←{⎕io ⎕ml←0 ·    ·    ·    ·    ·    ·    |cbst←{⎕io ⎕ml←0 ·    ·    ·    ·    ·    ·
·    'put'≡⍺⍺:⊃⍺ put ⍵    ·    ·    ·    ·    |·    'put'≡⍵:{⊃⍺ put ⍵}  ·    ·    ·    ·
·    'get'≡⍺⍺:⊃⌽⍵·get·⍺ 0·    ·    ·    ·    |·    'get'≡⍵:{⊃⌽⍵·get·⍺ 0}    ·    ·    ·
·    'rem'≡⍺⍺:⊃⍺ rem ⍵ 0 ·    ·    ·    ·    |·    'rem'≡⍵:{⊃⍺ rem ⍵ 0}·    ·    ·    ·
·    'fmt'≡⍺⍺:fmt·⍵  ·    ·    ·    ·    ·    |·    'fmt'≡⍵:fmt ·    ·    ·    ·    ·    ·
·    'vec'≡⍺⍺:vec·⍵  ·    ·    ·    ·    ·    |·    'vec'≡⍵:vec ·    ·    ·    ·    ·    ·
·    'chk'≡⍺⍺:0 chk ⍵·    ·    ·    ·    ·    |·    'chk'≡⍵:0 chk   ·    ·    ·    ·    ·
·    'bal'≡⍺⍺:bal·⍵  ·    ·    ·    ·    ·    |·    'bal'≡⍵:bal ·    ·    ·    ·    ·    ·

      ⍝ compare the two methods of function derivation:

      dput←'put'sbst                      ⍝ <trace> derivation of dput    ---.

      cput←cbst'put'                      ⍝ <trace> closure of cput       ---'

      dput                                ⍝ show derived put
 put  ∇sbst

      cput                                ⍝ show put closure
 {⊃⍺ put ⍵} ; ⎕ml←0, ⎕io←0, put, get, rem, rrot, search, fmt, vec, chk, bal, cmp
      , list

      cfmt←cbst'fmt'                      ⍝ generate a fmt function

      cfmt((tree cput 1)cput 3)cput 5     ⍝ tree with three more values
   .1=1
2=2|
   |   .3=3
   '4=4|
       '5=5

      dfmt tree dput 1                    ⍝ <trace> derived functions     ---.

      )sinl                               ⍝ show stack and local names       ·
sbst[34]*       wise    dir     inf     lft     rgt
sbst[104]       list    cmp     bal     chk     vec     fmt     search  rrot
        rem     get     put     ⎕IO     ⎕ML
   .1=1
2=2|
   '4=4

      cfmt tree cput 1                    ⍝ <trace> closures              ---'

      )sinl                               ⍝ show stack and local names
cbst[34]*       wise    dir     inf     lft     rgt
cbst[104]
;       list    cmp     bal     chk     vec     fmt     search  rrot    rem
        get     put     ⎕IO     ⎕ML
   .1=1
2=2|
   '4=4
```

```
    cmpx'((0 dput 1)dput 3)dput 5' '((0 cput 1)cput 3)cput 5'       ⍝ timings
((0 dput 1)dput 3)dput 5 1.7E¯4   0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
((0 cput 1)cput 3)cput 5 1.2E¯4 -30% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕

    ⍝ However, compare creation times:

    defd ← {dput←'put'sbst ⋄ ⍵}         ⍝ derive put function as side-effect
    defc ← {cput←cbst'put' ⋄ ⍵}         ⍝ generate put closure as side-effect

    cmpx'defd 0' 'defc 0'               ⍝ compare timings
defd 0 3.1E¯6    0% ⎕⎕⎕⎕⎕⎕
defc 0 1.9E¯5 +515% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕

    ⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝
    ⍝                                                           ⍝
    ⍝ We have seen:                                             ⍝
    ⍝                                                           ⍝
    ⍝   Left-argument binding (currying)  doub ← 2 ×            ⍝
    ⍝                                                           ⍝
    ⍝   Function results:   ·   ·   ·   · {⍵:⎕fstie ⋄ ⎕ftie}    ⍝
    ⍝                                                           ⍝
    ⍝   Parameterisation of functions:  · mul∆←{⍵×}             ⍝
    ⍝                                                           ⍝
    ⍝   Partial evaluation of functions:· cmix←{⎕ml←0 ⋄ ↑}0     ⍝
    ⍝                                                           ⍝
    ⍝   Persistent local variables: ·   · {N←⍵ ⋄ {(⍳N+←⍵)+N}}   ⍝
    ⍝                                                           ⍝
    ⍝   Shared local values:    ·   ·   · get←Q∆'get'           ⍝
    ⍝                                     put←Q∆'put'           ⍝
    ⍝                                                           ⍝
    ⍝   Full display of D-functions:·   · {⍵ ⍵}                 ⍝
    ⍝                                                           ⍝
    ⍝   Oh, and quieter system names:   · {⍵:⎕fstie ⋄ ⎕ftie}    ⍝
    ⍝                                                           ⍝
    ⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝⍝

    ⍝ Pros  :-)                         & Cons  :-(
    ⍝ ---------                         ⍝ ---------
    ⍝ Cute.                             ⍝
    ⍝                                   ⍝ Increased obfuscation opportunity.
    ⍝ Slicker than space-binding.       ⍝
    ⍝                                   ⍝ One more d**n thing.
    ⍝ Speed through partial evaluation. ⍝
    ⍝                                   ⍝ One less error condition.
    ⍝ Attractive to FP programmers.     ⍝
    ⍝                                   ⍝ Arguably, distorts the language:
                                         (refws'demo').food.chain
```

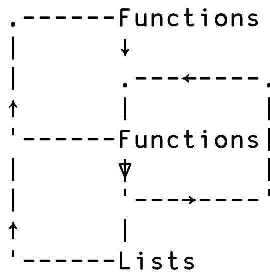
Food Chain: Classic APL




Food Chain: Function Results Edition




Food Chain: Typical Functional Programming Language (max.dws)




```
⍝ More thought required:
⍝ ---------------------
⍝ Default display of multi-line arrays and functions needs more work.
⍝
⍝ Decide on ⎕cr details.
⍝
⍝ A function-returning function applied by the primitive each operator
⍝ might return an array of functions:
⍝
⍝       {2|⍵:× ◇ ÷}¨ 1 2 3 4    ⍝ odd: × else ÷
⍝   × ÷ × ÷
⍝
⍝ WIBNI we could edit a closure with a tabbed edit window.
⍝ (this technique could also apply to editing derived functions).
⍝
⍝ For more on closures:     Google [Lexical Closures]

⍝   F:\Closures\demo.pdf        ⍝ Memory stick: see this session.
⍝   F:\Closures\setup.exe       ⍝ Memory stick: install this version,
⍝   )load demo                  ⍝               then try these examples.
```

In programming languages, a closure is a function that refers to free variables in its lexical context.

A closure typically comes about when one function is declared entirely within the body of another, and the inner function refers to local variables of the outer function. At run time, when the outer function executes, a closure is formed. It consists of the function code, and references to any variables in the outer function's scope that the closure needs.

Closures are commonly used in functional programming to defer calculation, to hide state, and as arguments to higher-order functions.

A closure combines the code of a function with a special lexical environment bound to that function (scope). Closure lexical variables differ from global variables in that they do not occupy the global variable namespace. They differ from object oriented member variables in that they are bound to function invocations, not object instances.

⍝ The End - Whad'ya think?