Chapter 1

Introduction to Daisy

Daisy is a list processing language with primitive operations for symbolic processing. Daisy is a language of expressions. Its interpreter reduces the graph representation of an expression to a value. If the interpreter reduces an expression E to a value v then v is said to be E's value.

1.1 Numerals and Arithmetic

The simplest expression is a number. An *integer* appears as a sequence of decimal digits, possibly with a minus-sign. Some examples are `597`, `-32`, and `0`. There is also a floating-point representation, which is indicated by a decimal point and possibly an exponent. Numerals `124.0` and `3.333333e-01` are both expressions of the same number.

The value of a number is that same number, up to equivalent spellings. For instance, `597` has value `597`; `0` has value `0`; and `124.0` has value `1.240000e+02`, the latter being the cannonical spelling used in output.

Daisy has a collection of arithmetic operations that apply to numbers One of these is `inc`, the increment-operation. The expression

```
inc:55
```

applies `inc` to the integer `55`. Its value is `56`, one more than `55`. Similarly, `inc:-3` has value `-2`. Incrementing does not alter the argument; `inc:`N creates a new number representing $N + 1$. Other numeric operations used in this section are

`dcr` Decrement; for instance, `dcr:5` has value `4` and `dcr:0` has value `-1`.

`neg` Negate; for instance, `neg:19` has value `-19`.

`add` Add; for instance, `add:[3 2]` yields the sum `5` of `3` and `2`.

`mpy` Multiply; for intance, `mpy:[3 2]` has value `6`, and `mpy:[-3 7]` has value `-21`.

In the examples above, an expression may appear wherever there is a number. The expression `inc:neg:19` yields `-18` because `inc` is applied to `-19`. Writing `add:[inc:5 neg:2]` is like writing `add:[6 -2]`; its value is `4`. Similarly, `mpy:[4 add:[dcr:3 17]]` yields `76`, representing $4 \cdot ((3-1)+17)$.

1.2 List expressions

The arguments above to `add` and `mpy`, forms like

```
[ 3  2 ]

[ inc:5  neg:2 ]

[ 4  add:[ dcr:3  17 ] ]
```

are *list expressions.* If $E_0, E_1, \ldots, E_n$ are Daisy expressions, then the expression

$$[\, E_0 \;\; E_1 \;\; \cdots \;\; E_n \,]$$

has the list value

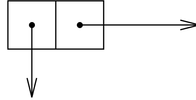
$$[\, v_0 \;\; v_1 \;\; \cdots \;\; v_n \,] \quad ,$$

where each v_i is the value of E_i. The empty list `[ ]` has value `[ ]` and is called *Nil.* The list expression `[1 23 -7]` yields `[1 23 -7]` because each of the element-expressions is a number. Here are more examples:

Expression	Value
`[ 3  2 ]`	`[ 3  2 ]`
`[ inc:5  neg:2 ]`	`[ 6  -2 ]`
`[ dcr:3  17 ]`	`[ 2  17 ]`
`[ 4  add:[ dcr:3  17 ] ]`	`[ 4  19 ]`
`[ inc:2  neg:2 ]`	`[ 3  -2 ]`
`[ 12  [inc:2  neg:2]  99 ]`	`[ 12  [ 3  -2 ]  -5 ]`

Lists are represented as binary records whose two fields are its *head* and

its *tail.* These are citations (or "pointers") to other objects:



List representation is discussed further in later sections. The separator '`!`' is used when a list's tail is explicitly given.

Expression	Value
`[ 5 ! inc:neg:6 ]`	`[ 5 ! -5 ]`
`[ [inc:2  neg:2] ! 99 ]`	`[ [ 3  -2 ] ! 99 ]`

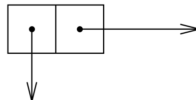
The list expression

$$[\,E\ !\ E'\,]$$

yields the value

$$[\,v\ !\ v'\,]$$

of values for E and E'. That is, $[\,E\ !\ E'\,]$ expresses the cell



Daisy's parsing and display primitives suppress cascading dot-notation when lists' tails are lists. Suppose U, V, and W are expressions whose values are u, v, and w respectively.

Expression	Value	Appears as
$[\,]$	$[\,]$	$[\,]$
$[\,W\,]$	$[\,w\ !\ [\,]\,]$	$[\,w\,]$
$[\,V\ \ W\,]$	$[\,v\ !\ [\,w\ !\ [\,]\,]\,]$	$[\,v\ \ w\,]$
$[\,U\ \ V\ \ W\,]$	$[\,u\ !\ [\,v\ !\ [\,w\ !\ [\,]\,]\,]\,]$	$[\,u\ \ v\ \ w\,]$
$\vdots$	$\vdots$	$\vdots$

The same convention is used in list expressions. For $[\,U\ \ V\ \ W\,]$, one may write

$$[\,U\ !\ [\,V\ !\ [\,W\ !\ [\,]\,]\,]\,]$$

1.3 Literal Symbols

A *literal* is a symbol, such as `train` or `orange`. Like numbers, literals are atomic data; they have no structure. A sequence of characters surrounded by double-quotes forms a *literal quotation*, whose value is the literal with the same spelling. For instance, the expression

```
"train"
```

has value

```
train
```

Similarly,

Expression	Value
"cars"	cars
"trucks"	trucks
"things that go"	things␣that␣go
["things that go"]	[things that go]
["things" "that" "go"]	[things that go]

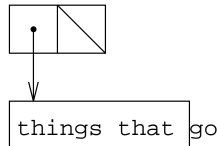
The third example above shows the space character incorporated in the literal's spelling. The '␣' does not appear when the name is displayed. Though `things␣that␣go` appears in three parts because of the spaces, it is an atomic symbol. The list expressions

```
["things that go"]
```

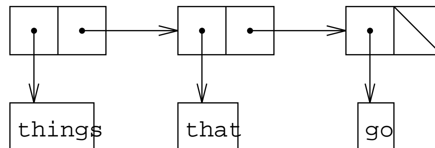
and

```
["things" "that" "go"]
```

have values that look the same, but these are different structures: the first is a one-element list



and the second is a three-element list



1.4 Definitions

In mathematical notation, one might define the multiplication of a pair by a scalar as

$$s \otimes [n_1, n_2] \stackrel{\text{def}}{=} [s \cdot n_1,\ s \cdot n_2]$$

This formula says what the symbol '$\otimes$' stands for by showing its function on a configuration of variables. An analogous Daisy definition is expressed like this:

```
ScalePair =
    \[S [N1 N2]] . [ mpy:[S N1]  mpy:[S N2] ]
```

If I is a literal and E is an expression, the form

$$I \boxed{=} E$$

assigns E's value to the name I. The name `ScalePair` plays the role of the symbol '$\otimes$'; Daisy has no provisions for infix notation or special characters. The form

$$\boxed{\backslash} X \boxed{.} E$$

is called a *function expression*. Assigning a function to a name is like making a procedure definition. X is a *formal argument* (or parameter) and the *body* E may be any expression. `ScalePair`'s formal argument is `[S [N1 N2]]`. When `ScalePair` is applied, `S`, `N1`, and `N2` are *bound to* (or associated with) the corresponding components of the argument. The function's body is evaluated as though occurances of `S`, `N1`, and `N2` had been replaced by their bindings. For instance, `ScalePair:[3 [-2 7]]` binds `S` to `3`, `N1` to `-2`, and `N2` to `7`. It is as though `ScalePair`'s body had become

```
[ mpy:[3 -2] mpy:[3 7] ]  .
```

Recall that `mpy` is the multiplication operation, so the resulting value is `[-6 21]`.

The textual substitution of pieces of an actual argument for corresponding identifiers is called *symbolic expansion*. The Daisy interpreter does not actually work this way; instead, it develops bindings in a data structure. Even so, the resulting value is almost always the same, and one often reasons about programs in this manner. Here is a sketch of how `ScalePair:[3 [-2 7]]` is actually interpreted.

1. The literal `ScalePair` is evaluated. Since it has been assigned, its value is the function expressed as

   ```
   \[S [N1 N2]] . [ mpy:[S N1]  mpy:[S N2] ]
   ```

2. The argument `[3 [-2 7]]` is evaluated to yield `[3 [-2 7]]`.

3. The formal argument `[S [N1 N2]]` is bound to the actual argument `[3 [-2 7]]`. The association is retained in a data structure called an *environment*.

4. The expression `[mpy:[S V1] mpy:[S V2]]` is interpreted. Occurances `mpy` are resolved to the primitive multiply-operation. Occurances of `S`, `V1`, `V2` are resolved their bindings as established in step (3). This yields the result `[-6 21]`.

1.5 Recursive programs

A more general function for scalar multiplication applies to vectors of arbitrary length:

$$s \otimes [n_1, n_2, \ldots, n_n] \stackrel{\text{def}}{=} [s \cdot n_1,\ s \cdot n_2,\ \ldots,\ s \cdot n_n]$$

A Daisy version is

```
ScaleVector = \[S V] .
   if:[ nil?:V
         []
         [ mpy:[S head:V]  !  ScaleVector:[S tail:V] ]
      ]
```

The `nil?` operation tests whether its argument is the empty list; `head` and `tail` retreive the fields of a list. To understand the Daisy definition, one must know that vectors are represented as lists of numbers. This was more explicit in the `ScalePair` definition. `ScaleVector` multiplies each number in `V` by the scalar quantity `S`, producing a list of products, hence another vector.

The `if` is Daisy's conditional operation.

$$\texttt{if:[}p\ u\ v\texttt{]}$$

returns u where p is true and v where p is false. In `ScaleVector`, p is the value of the expression `nil?:V`; u is the value of `[]`; and v is the value of the list expression

```
[ mpy:[S head:V]  !  ScaleVector:[S tail:V] ]
```

Symbolically expanded, `ScaleVector:[3 []]` is

```
if:[ nil?:[]
       []
       [ mpy:[3 head:[]]  !  ScaleVector:[3 tail:[]] ]
   ]
```

The expression `nil?:[]` has value `T`, the result of a true test. Hence, the conditional expression reduces to

```
if:[T [ ] [? ! ?]]
```

and `if` returns `[]`. Since the test was true, it doesn't matter what the second alternative value is.

The second alternative in `ScaleVector`'s body involves a recursive call; `ScaleVector:[3 [2]]` expands to

```
if:[ nil?:[2]
       []
       [ mpy:[3 head:[2]]  !  ScaleVector:[3 tail:[2]] ]
   ]
```

Since `[2]` is not Nil, `if` chooses

```
[ mpy:[3 head:[2]]  !  ScaleVector:[3 tail:[2]] ]
```

which simplifies to

```
[ mpy:[3 2]  !  ScaleVector:[3 []] ]
```

This list's head has value `6`, and its tail has value `[]`, as was just shown. Hence, the result is the list `[6 ! []]`, or `[6]` according to the '`!`' convention. Similarly, `ScaleVector:[3 [1 2]]` expands to

```
[ mpy:[3 1]  !  ScaleVector:[3 [2]] ]
```

and simplifies to `[3 6]`. `ScaleVector:[3 [0 1 2]]` yields `[0 3 6]`.
And so on. The recursion in this example is achieved through assignment, whose global effect makes the name `ScaleVector` an indirect reference to a program, even within that program's body.

1.6 Binding Forms

An equivalent definition of `ScaleVector`, below, uses the binding form `let` to give names to `V`'s head and tail.

```
ScaleVector = \[S V] .
  let:[ [N ! Ns]
        V
        if:[ nil?:V
              []
              [ mpy:[S N] ! ScaleVector:[S Ns] ]
           ]
      ]
```

The form

$$\texttt{let:[}X\ \ E\ \ E'\texttt{]}$$

binds the formal argument X to the value of E for an evaluation of expression E'. The brackets '`[`' and '`]`' must be used. In `ScaleVector`,

X is `[N ! Ns]`

E is `V`

E' is
```
if:[ nil?:V
      []
      [ mpy:[S N] ! ScaleVector:[S Ns] ]
   ]
```

Binding `[N ! Ns]` to `V` associates the identifier `N` with `V`'s head and `Ns` with `V`'s tail. It is as though one were saying "Wherever you see '`N`', read: '`head:V`'." Though they are not exactly equivalent, this use of `let` is like the auxiliary definition of `Phantom` below.

```
ScaleVector = \[S V].
   if:[ nil?:V
         []
         Phantom:[S V]
      ]

Phantom = \[S [N ! Ns]].
   [ mpy:[S N] ! ScaleVector:[S Ns] ]
```

`Phantom` just serves to identify the components of `V`. The `let` makes the same association locally.

A `let`-expression develops bindings from the surrounding scope; The form

$$\texttt{rec:[}X\ \ E\ \ E'\texttt{]}$$

develops recursive bindings for the identifiers in X. As with `let`, it is important that `rec`'s argument be surrounded with brackets, not angles. The

variation of `ScaleVector` below locally defines a recursive procedure, `LOOP`, that multiplies each element of a list by `S`.

```
ScaleVector = \[S V] .
  rec:[ LOOP
        \[N ! Ns] .
           [ mpy:[S N] ! if:[ nil?:Ns [] LOOP:Ns ] ]

        if:[ nil?:V [] LOOP:V ]
      ]
```

A `nil?`-test guards each call to `LOOP`, so it is never applied to an empty vector. The duplicate occurances of

```
if:[ nil?:⋆ [] LOOP:⋆ ]
```

could be abstracted to a function:

```
ScaleVector = \[S V] .                          |
  rec:[ [ LOOP  HELP ]                          |
        [ \[N ! Ns] . [mpy:[S N] ! HELP:Ns] | LOOP defn
          \Ns . if:[ nil?:Ns  []  LOOP:Ns ] | HELP defn
         ]                                      |
        HELP:V                                  |
       ]
```

which function goes with which name. Both `let` and `rec` group formal parameters their binding expressions seperately. This version of `ScaleVector` recursively binds `[LOOP HELP]` to a pair of function expressions `[` L H `]`; it could be *read* as a mutually recursive system of definitions,

```
LOOP = \[N ! Ns] . [ mpy:[S N] ! HELP:Ns ]
HELP = \Ns . if:[ nil?:Ns  []  LOOP:Ns ]
```

However, this system of equations is not the same as a pair of surface assignments. First, the occurance of `S` in `LOOP`'s definition acquires its binding from `ScaleVector`'s formal argument. Second, the names `LOOP` and `HELP` retain their values only in the scope of the `rec` expression.

1.7 Function values

The locally defined `LOOP`, just above, maps a multiplication operation over a list of numbers. Generalizations of such functions are useful. At the same

time, a multiplication-mapping function is not a very broad generalization, although it might be contemplated anyway. A more general mapping is developed in two steps below. The first is to identify as `Scale` a *multiply-by-*`S` function.

```
ScaleVector = \[S V] .
  let:[ Scale
        \M . mpy:[S M]

        rec:[ [ LOOP  HELP ]
              [ \[N ! Ns]. [ Scale:N ! HELP:Ns ]
                \Ns. if:[ nil?:Ns  []  LOOP:Ns ]
               ]
              HELP:V
             ]
       ]
```

The `Scale` function depends on a binding for `S`. Inheritence of bindings is lexical. This means that identifiers capture bindings according to language structure. The '`S`' in `\M.mpy:[S M]` captures its binding from `ScaleVector`'s formal argument, `[S V]`, because that is the closest point—in a hierarchical sense—that `S` occurs in a formal argument. Similarly, the '`Scale`' in `LOOP` gets its value from the surrounding `let` binding.

`LOOP` maps `Scale` over the list `V`. The desired generalization, call it `MapVector`, comes in moving `LOOP` outside `ScaleVector`.

```
ScaleVector = \[S V] .
   let:[ Scale
         \X . mpy:[S X]
         MapVector:[ Scale V ]
        ]

MapVector = \[F V] .
   rec:[ [ LOOP  HELP ]
         [ \[N ! Ns] .  [ F:N ! HELP:Ns ]
           \Ns . if:[ nil?:Ns  []  LOOP:Ns ]
          ]
         HELP:V
        ]
```

Since `MapVector` is outside `ScaleVector`'s scope, `Scale` must now be passed as an argument. The only purpose of the `let` in `ScaleVector` is to name

the once-occuring *multiply-by-*`S` function. One would likely omit it and instead define

```
ScaleVector = \[S V]. MapVector:[ (\N.mpy:[S N])  V ]
```

The parentheses around the function expression `(\N.mpy[S N])` are for emphasis and are not required. Until this example, function expressions have occured at "special" places in defining and binding forms. However, functions may appear anywhere; and in particular, in argument lists.

Now suppose that a *vector* is represented as a list of number-literal pairs, such as

```
[ [5 trucks] [2 cars] [0 boats] [1 airplanes] [9 taxis] ]
```

Define a function that translates numbers to literals:

```
HowMany = \M. \[N T].
   if:[ eq?:[N 0] ["no" T]
        eq?:[N 1] ["one" T]
        eq?:[N 2] ["a couple of" T]
        lt?:[N M] ["a few" T]
                  ["many" T]
      ]
```

The new primitives are numeric tests: `eq?` tests for equality and `lt?` is a less-than test. The `if` operation allows successive tests.

$$\texttt{if:[}\, p_0 \;\; v_0 \quad p_1 \;\; v_1 \quad \cdots \quad p_n \;\; v_n \quad v_{n+1} \,\texttt{]}$$

returns the first v_i whose test value p_i is true. Should none of the tests be true, `if` returns v_{n+1}.

`HowMany` has the form `\M.\[N T].`E. It takes a number, `M` and returns a function from pairs to pairs. For instance, `HowMany:5` symbolically expands to a *function* giving an English description of size depending on the inherited binding for the number `M`—`5` in this case. So the argument `5`, in effect, specializes `HowMany` to the function shown below.

```
\[N T]. if:[ ··· lt?:[N 5] ["a few" T] ··· ]
```

Thus,

`(HowMany:5):[3 "cars"]` has value `[a few cars]` and

`(HowMany:5):[5 "trains"]` has value `[many trains]`

The parentheses above are needed because without them, application would be done from right to left. That is, the form $F : G : X$ applies F to $G : X$, according to the default precedence. The revision to `ScaleVector` is

```
ScaleVector = \[S V] .  MapVector:[ HowMany:S  V ]
```

`MapVector` is unchanged. Under these definitions,

```
ScaleVector:[ 6 [ [5 "trucks"] [2 "cars"] [0 "boats"]
                  [1 "airplanes"] [9 "taxis"]         ]
            ]
```

evaluates to

```
[ [a few trucks]  [a couple of cars]  [no boats]
  [one airplanes]  [many taxis]
 ]
```

1.8 Data recursion

Let us consider one last generalization of scalar multiplication, to infinite sequences:

$$s \otimes [n_1, n_2, \cdots] \stackrel{\text{def}}{=} [s \cdot n_1, \ s \cdot n_2, \ \cdots]$$

The `ScaleVector` function already implements this specification, and infinite sequences can be represented in Daisy. For example, a list of increasing integers, `[0 1 2` $\cdots$ `]` can be developed in two ways. A recursive function definition,

```
Integers = \N .  rec:[ F
                       \M . [ M ! F:inc:M ]
                       F:N
                      ]
```

defines `F` to concatenate `M` to the list of all its successors (Recall that `inc` is the increment-operation.):

```
Integers:0     = F:0
               = [0 ! F:inc:0]
               = [0 ! F:1]
               = [0 ! [1 ! F:inc:1 ] ]
               = [0 1 ! F:2 ]
               = [0 1 ! [2 ! F:inc:2 ] ]
               = [0 1 2 ! F:3 ]
                 ⋮
               = [0 1 2 ···]
whose value is = [0 1 2 ···]
```

The same result is obtained if, instead of defining a function recursively, the list itself is recursively defined. Below, the list L is described in terms of itself.

```
Integers = \N .
  rec:[ Z
        [ N ! MapVector:[inc Z] ]
        Z
       ]
```

MapVector is the same as before; it applies inc to each element of the list Z. Reasoning symbolically,

```
Integers:0 = Z
  = [0 !  MapVector:[inc Z] ]
  = [0 ! MapVector:[inc [0 ··· ] ] ]
  = [0 ! [1 ! MapVector:[inc tail:[0 ! [1 ··· ]] ] ]]
  = [0 1 ! MapVector:[ inc [1 ··· ] ] ]
  = [0 1 ! [2 ! MapVector:[ inc tail:[1 ! [2 ··· ]] ] ]]
  = [0 1 2 ! MapVector:[ inc [2 ··· ] ] ]
   ⋮
  = [0 1 2 ··· ]
 whose value is
    [0 1 2  ··· ]
```

The expression ScaleVector:[3 Integers:0] yields the list

```
[ 0 3 6 9 12 15 18 21 ··· ]  .
```

Should it be used *only* on nonterminating lists, MapVector could be streamlined by omitting its termination test.

```
MapVector = \[F Ns] .
  rec:[ LOOP
        \[N ! Ns] . [ F:N ! LOOP:Ns ]
        LOOP:Ns
       ]
```

1.9 Daisy's Construction Functionals

There is a primitive operation in Daisy that does the work of MapVector: map takes a function F an returns a *function* which maps F over a list. A function which takes functions to functions is called a *functional*. The map operation can be specified by a Daisy definition:

```
map = \F.
 rec:[ LOOP
       \L.
         let:[[H ! T] L
          if:[ nil?:L
               []
               [F:H ! LOOP:T]
              ]]
       LOOP
      ]
```

LOOP is defined within the scope of map's function binding, so the value of F is determined once map is applied to something. Then, where (map:F) is applied to a list LOOP does the mapping. Of course, the primitive map operation runs more efficiently, but its general behavior is identical.

The version of ScaleVector shown below binds F to a multiplier that takes its arguments one at a time, a transformation known as *currying*. Thus, \F:S, a *multiply-by-*S function, is mapped over the sequence of values, Vs.

```
ScaleVector = \[S Vs] .
  let:[ F
        \X. \Y. mpy:[X Y]
         (map:(F:S)):Vs
       ]
```

Daisy has a number of other functionals for mapping functions over structures. The most general of these is called fc. It applies takes a list of functions to an "array of arguments." An expression of the form

$$(\mathrm{fc}\texttt{:[}F\ \texttt{!}\ F'\texttt{])}\ \texttt{:}\quad \begin{array}{l}\texttt{[}\ \texttt{[}E_0\ \texttt{!}\ E'_0\texttt{]}\\ \quad\texttt{[}E_1\ \texttt{!}\ E'_1\texttt{]}\\ \qquad\vdots\\ \texttt{]}\end{array}$$

is like the expression

$$\texttt{[}\ F\texttt{:[}E_0\ E_1\ \cdots\texttt{]}\ \texttt{!}\ \texttt{(fc:}F'\texttt{):[}E'_0\ E'_1\ \cdots\texttt{]]}$$

Here are some examples:

This expression	is like	Value
`(fc:[add add]):[ [ 1 2 ] [ 3 4 ] ]`	`[ add:[1 3] add:[2 4] ]`	`[4 6]`
`(fc:[mpy mpy add]):[ [ 1 2 3 ] [ 4 5 6 ] ]`	`[ mpy:[1 4] mpy:[2 5] add:[3 6] ]`	`[4 10 9]`

The following version of `ScaleVector` uses `fc` to map a sequence of `mpy`s

across two sequences of numbers:

```
ScaleVector = \[S V] .
   rec:[ Ss [ S  ! Ss]
   rec:[ Ms [ mpy  ! Ms]
         (fc:mpy):[ Ss V ]
       ]]
```

The identifier `Ms` is recursively bound to the list `[mpy !  Ms]`, making `[mpy mpy ··· ]`. Similarly, `Ss` expands to `[S S ··· ]`. Hence, an expansion of the expression `ScaleVector:[3 [0 1 2]]` gives

```
[ mpy  mpy  mpy ···]:[
[  0    1    2  ]
[  3    3    3  ···] ]
```

The `fc` functional develops an application along each column, and also provides an implicit `nil?`-test for termination. The form above is essentially

```
[ mpy:[0 3]  mpy:[1 3]  mpy:[2 3] ]
```

with the result `[0 3 6]`, as wanted.

This `fc` version of `ScaleVector` is the most efficient of all the versions because its execution does not involve the repeated creation of function environments.[1]. For this reason and others, the use of data recursion and construction functionals is prevalent in Daisy programming.

[1] However, if a curried version of `mpy` were provided as a primitive in Daisy, the version of `ScaleVector` using `map` would be best of all

1.10 Indeterminate Lists

An expression of the form

$$\texttt{set:[}\, E_0 \;\; E_1 \;\; \cdots \;\; E_n \,\texttt{]}$$

evaluates to a list

$$[\, v_{0'} \;\; v_{1'} \;\; \cdots \;\; v_{n'} \,]$$

where each $v_{i'}$ is the value of *one of the* E's. That is, the `set`-expression specifies the elements of a list but not their order. The ordering is determined through concurrent evaluation of the individual elements. The earlier an element-expression yields a value, the earlier that value appears in the resultant ordering. The expression

```
set:[ add:[2 3]  "truck"  inc:5 ]
```

yields one of the values `[5 truck 6]`, `[truck 5 6]`, `[6 5 truck]`, `[5 6 truck]`, `[truck 6 5]`, or `[6 truck 5]`. The most likely result is `[truck 6 5]`. Evaluating `"truck"` is easiest; evaluating `add:[2 3]` is hardest; and the differences are significant.

However, the ordering is unpredictable when comparable, non-trivial expressions are involved. The following function returns the message `Msg` after a count-down.

```
DELAY = \[N Msg] .  if:[ zero?:N
                             Msg
                             DELAY:[dcr:N Msg]
                         ]
```

`Zero?` is a test-for-0, and `dcr` is the decrement-operation. Successive evaluations of the expression

```
set:[ DELAY:[100 "A"]  DELAY:[100 "B"]  DELAY:[100 "C"] ]
```

yield various permutations of `A`, `B`, and `C`.

The following variation of the `HowMany` function, introduced earlier, gives some uncertainty to its outcome:

```
HowMany = \M. \[N T].
   rec:[ GUARD
         \[P V] . if:[ P V GUARD:[P V] ]

         head:set:[
             GUARD:[eq?:[N 0]  ["no" T] ]
```

```
                GUARD:[eq?:[N 1]  ["one" T] ]
                GUARD:[eq?:[N 2]  ["a couple of" T] ]
                DELAY:[add:[-2 N] ["a few" T] ]
                DELAY:[M          ["many" T] ]
              ]
            ]
```

`HowMany`
returns the head of an concurrently ordered list; it chooses from five alternative descriptions of `N`. Unlike before, it is indeterminate which alternatives might be obtained for certain values of `N`. The function `GUARD` returns a value `V` provided its test `P` is true; otherwise, `GUARD` loops forever and thus fails to return anything. Should `GUARD` fail to return a value, that alternative cannot be choosen by `HowMany`. For instance, when `N` is `1`, the choice `"a couple of"` is impossible. With this new definition, and with `ScaleVector` defined as

```
ScaleVector = \[S V] .  MapVector:[ HowMany:S  V ]
```

the expression

```
ScaleVector:[ 6 [ [5 "trucks"] [2 "cars"] [0 "boats"]
                  [1 "airplanes"] [9 taxis]
                ]
            ]
```

produced the following values on six sucessive attempts

```
[ [a few trucks] [a couple of cars] [no boats]
  [one airplanes] [a few taxis] ]

[ [a few trucks] [a couple of cars] [no boats]
  [many airplanes] [a few taxis] ]

[ [a few trucks] [a couple of cars] [no boats]
  [many airplanes] [a few taxis] ]

[ [a few trucks] [a few cars] [no boats]
  [one airplanes] [many taxis] ]

[ [a few trucks] [many cars] [no boats]
  [one airplanes] [many taxis] ]

[ [many trucks] [a couple of cars] [no boats]
  [one airplanes] [many taxis] ]
```

Chapter 2

Using Daisy

A single display character is sometimes placed in a box for emphasis. When it is necessary to show non-graphical control codes, it is done by placing a mnemonic for the code in the box. The mnemonics used most frequently are

EOT	end-of-transmission (control 'D')
ETX	host interrupt (control 'C')
BEL	audible character (control 'G')
NL	new line (control 'J' in UNIX)

The true location of the Daisy program varies with different instalations. The name `Daisy` may be an alias for that object, or it may be located in a default directory-search path. In UNIX systems, aliasing is accomplished by a command of the form

`alias Daisy` *DaisyObject*

where *DaisyObject* is the true directory location.

The `Daisy` command has six optional arguments. Of these, '`m`' and '`i`' are generally useful, and '`s`' is sometimes used to explore concurrent programs. The options '`n`', '`t`', and '`f`' are used in development.

`-m` This option sets the bound on the list space. Is default value is 100,000. Entering

```
Daisy -m 500000
```

directs DSI—Daisy's underlying list processing system—to allocate 500,000 cells of heap space. If the number entered exceeds the amount of physical memory available, DSI allocates as much as it can.

`-i` This option names a source file to be taken as input before interaction begins. The default is a fixed file called `Daisy.d`. Announcements are sometimes posted on this file. Typing

`Daisy -i` *FileName*

initially directs input to the file *FileName.*

`-s` This option sets the upper bound on the multitasking granularity. It should be followed by a number beteen 1 and 255.

`-n` This option sets the lower bound on the multitasking granularity. It should be followed by a number beteen 1 and 255.

`-t` This option sets levels and regions of implementation tracing. It is not used in programming

`-f` This option names a file for tracing and diagnostic output. It is not used in programming

2.1 Invoking Daisy

Below is the display of an interactive session with Daisy. System utterences are shaded, and new-line characters occur as indicated by line breaks.

```
▮ Daisy

Daisy (DSI V4R0(beta)) 5/26/92.
[#]
& EOT [#]
DSI(002) 11/4/86
▮
```

The symbol ▮ stands for the host's prompt; the `Daisy` command invokes Daisy. This program first issues a banner stating its version and release date. The first value displayed is `[#]`; it is the result of reading an empty file. The '`-i`' option is used to redirect initial input to a host file. Suppose that the file `greetings` contains the text

```
"Hello, there."
```

Directing initial input to this file is done like this:

```
▆ Daisy greetings

Daisy (DSI V4R0(beta)) 5/26/92.
[Hello, there]
& EOT [#]
DSI(002) 11/4/86
▆
&
```

The ampersand, '&', is Daisy's prompt. Once this prompt is raised, interaction with the Daisy interpreter is in progress. In both of the sessions above, the operator types the EOT character in response to the first prompt. Daisy displays the list [#] and terminates.

The interactive interpreter produces a list of values corresponding to the expressions entered by the operator. This list is displayed on the operator's terminal screen, along with prompts and echoed input. The EOT character terminates input, the interpreter's response is [#] because no expressions are entered by the operator. The one-element list [#]—one might expect []—is due to properties of Daisy's scanning and parsing primitives. Here is another, longer Daisy session.

```
▆ Daisy

DSI(002) 11/4/86
[#]
& 11
[11
& +121 -123321
 121 -123321
& "bus"
 bus
& ["bus" 5]"
 [bus 5]
& EOT ]
DSI(002) 11/4/86
▆
```

Let us again dissect the terminal's display; it has three interleaved parts. The operator has typed a sequence of expressions. These are incorporated

in a list,

[11 NL +121 -123321 NL "bus" NL ["bus" 5] NL] .

There is no difference in representation between this "top level" list and list expression `["bus" 5]` it contains. Evaluation of this list's elements yields

[11 NL +121 -123321 NL bus NL [bus 5] NL] .

Finally, there is a sequence of prompts raised by the terminal handler when input is expected.

These distinct character streams appear concurrently on the terminal screen. In particular, the first character issued by interactive Daisy is a '`[`' after the operator's first new-line—it opens the list of interpreted values—and the last character it issues is a closing '`]`'.

2.2 Getting Out of Daisy

To terminate an interactive session with Daisy, one normally types an end-of-transmission character in response to Daisy's prompt. This is illustrated in the first examples above. Abnormal termination typically occurs in one of three ways. Daisy can exhaust its space of available cells; interpretation may *diverge*, or fall into a loop; the operator may displayed an nonterminating answer.

In the fragment below, the operator asks for the value of "X, where $X = X + 1$." This is a divergent expression.

```
& rec:[ X inc:X X ]
BEL DSI abort; GC -- no free space
Memory Dump ['y']?
Exit
DSI(002) 11/4/86
```

To the memory-dump query, the operator types a new-line (or anything but '`y`') to say "No." The DSI system issues a BEL character each time it invokes storage reclamation. On most terminals, this character produces a sound. Below, the operator tests whether the literal `X` is an element of the

nonterminating list [0 1 2 ···].

```
& in?:[ "X"
&       rec:[ N
&             [0 !  [add*]:[[1*] N]]
&             N ]
&     ]
BEL  BEL  BEL  BEL  ETX
█
```

The storage consumed in interpretating this divergent expression is all recycled. The succession of BEL signals would continue indefinately, but the operator types the host's interrupt, ETX, to kill the Daisy program. Control resumes at the host command level.

There are divergent expressions that consume no storage. An example is in?:["X" [0*]], which tests for the presence of the literal X in a cyclic list of zeros.

```
& in?:[ "X"  [0 *] ]
—nothing–happens—  ETX
█
```

Interruption is also the only recourse when Daisy is asked to display a nonterminating structure. Below, the operator asks for an unending list of zeros and ones, and must interrupt to regain control.

```
& rec:[ L [0 1 !  L] L ]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
 0 1 0 1 0  ETX
█
```

2.3 Input and Output

The Daisy expression

evlst:prsis:scnis:dski:*File*

reads a Daisy source file. *File* is a literal that spells the name of a host text file. Suppose the file `defns` contains the following text.

```
idy = ^\x.x          | An identity function
xps = [idy *]        | Transposes rectangular structure
                     |
EX0 = ^[[a b c]      | For testing.
        [d e f]]     |
```

Below, this file is read by the operator.

```
▘ Daisy

DSI(002) 11/4/86
[#]
& evlst:xparses:scans:dski:"defns"
[[idy
 xps
 EX0
]
& EX0
 [[a b c] [d e f]]
& xps:EX0
 [[a d] [b e] [c f]]
&
```

The result of the first expression, the list

[idy NL xps NL EX0 NL] ,

reflects three assignments—the value of an assignment command being literal assigned. The following file, call it `initial`, is popular.

```
Load = \File. evlst:pris:scnis:dski:File
```

This defines `Load` to be a function that loads a Daisy file; thus, it saves a little typing:

```
% Daisy -i initial

DSI(002) 11/4/86
[Load]
& Load:"defns"
[[idy
 xps
 EXO
]
&
```

In Daisy's logical view, a file is a list. Suppose the file `showme` contains the following text.

```
[16 ["red" "trucks"]]
```

The `dski` operation creates a list of characters corresponding to the text in a host file.

```
&dski:"showme"
 [[ 1 ␣ 6 ␣ [ " r e d " ␣ " t r u c k s " ] ]
]
```

Each element of this list is a literal atom, with a single-character display name. Included in the list are any spaces and new-lines that occur in the `showme` file. The space characters, '␣' don't appear in the actual display. The '`1`' and '`6`' are literal characters, not numerals.

The `scans` operation builds a list of atoms and literal quotations from a list of characters. For instance, the character sequence `[` ··· `"` `r` `e` `d` `"` ··· `]` is incorporated as a quotation `[` ··· `"red"` ··· `]`; and `[` ··· `1` `6` ··· `]` is incorporated as the numeral `[` ··· `16` ··· `]`.

```
& scans:dski:"showme"
 [[ 16 [ "red" "trucks" ] ]
]
```

Other elements of the atom-list are character symbols `[` and `]` and NL, of meaning in parsing. The resulting list, then, is

```
[ [ 16 [ "red" "trucks" ] ] NL ]
```

The `xparses` operation builds a list of expressions from the atoms:

```
& xparses:scans:dski:"showme"
 [[16 ["red" "trucks"]]
]
```

The result above is a two-element list, containing a list expression and a new-line character. The `evlst` operation takes an expression-list and returns of list of values.

```
& evlst:xparses:scans:dski:"showme"
 [[16 [red trucks]]
]
```

The value list also includes a new-line.

The `console` operation establishes a kind of channel to the operator's key board, producing a list of literal characters corresponding to the operator's key strokes. In other respects, it is just like `dski`. Its argument is a literal prompt, which is displayed when input is expected. Usually, once the prompt is raised, the host retains control until the next new-line is entered. `Console` terminates its list where the operator enters an EOT. In the following fragment, `xparses` develops a sequence of atoms from key board input.

```
& xparses:scans:console:"??  "
[??  She tried and she tried
She tried and she tried
??  but her wheels would not turn
 but her wheels would not turn
??  EOT]
```

The `dsko` operation places text on a host file. It expects a list `[`*FileName Characters*`]`, whose first element, a literal, names the output file and whose second element, a list of characters, is the text to be placed there. The expression

```
dsko:["there" ["A" "B" "D"]]
```

has places the text `ABC` on a file named `there`. The value `[]` is returned *after all the text has been written,* at which time, the file is also closed.

The `screen` operation is like `dsko`, except that it displays text on the operator's terminal. For instance,

```
& screen:["A" "B" "C"]
 ABC[]
&
```

Here, `screen` displays the text `ABC` and then returns `[]`.

The text arguments to `dsko` and `screen` must be character lists, such as those created by `dski` and `console`. The `issue` operation takes an arbitrary structure and expands it to a stream of characters.

```
& issue:["track" 17]
 [[ t r a c k ␣ 1 7 ]]
&
```

`Issue`'s result is the list

[[t r a c k ␣ 1 7]] ,

including the list delimiter characters '`[`' and '`]`'.

Beginning programmers often resort to `screen` and `dsko` to achieve output. However, since file output and screen display are effects, their casual use is not regarded as applicative style. At the same time, good applicative methods for input and output are not well understood and continue to be a subject of language research. The example below is not offered as an example of good style. The expression entered by the operator creates two concurrent instances of output to the screen.

```
& Daisy -s 2
[#]
[& set:[screen:issue:"IS THERE AN ECHO IN HERE?"
&   screen:console:"??"]
 [??hello
IhSe lTlHoE
??out
RoEut
A??there
tN heECreHO
??[EOT] IN HERE?[] []]
```

The command argument "`-s 2`" lowers the multitasking granularity, in order to exaggerate the effect. Two applications of `screen` are placed in a multiset. The *value* of the expression is the list `[[] []]`, but neither instance of screen returns its `[]` until it has displayed all its text. The first `screen` displays the characters of a literal produced by `issue`. The second `screen` echos console input to the prompt "`??`". The display effects overlap as an order is developed for the multiset.

About the same effect would appear on a file if `dsko` were used in place of `screen`.

2.4 Errors

Below, the operator enters an erroneous expression

```
& taxicab
|ubi:taxicab|
&
```

What appears as |ubi:taxicab| is an error-value, or *erron*. It says, "The identifier taxicab is unbound." The occurance of an error does not trap to top level or anything like that; the erron merely records the occurance. In the fragment below, an error occurs within a list expression.

```
& [ inc:N "cars"]
[|nn0/ubi:N| cars]
&
```

The prefix 'nn0' says "A nonnumeric operand;" inc expects a numeral. The '/' might be read as "due to," so the whole message is "A nonnumeric operand, due to an unbound identifier, N." Error messages are built from a fixed set of message fragments, explained in [the section about *Errons*].

Daisy's parsing operation develops its own system of error messages. These are abbreviated explanations of where parsing fails. The examples below illustrate some effects of syntax errors during interaction.

```
& [1 2 3]
|val/[nnn@`]'|
& [A.B]
|val/..[i@`.'| |ubi:B|
|val/@`]'|
&
```

Here is what happens: first, parsing builds a list from the key board input. The list, if displayed, would appear as

```
[ |<nnn@`]'|  NL  |..[i@`.'|  B  |@`]'|  NL  ··· ]
```

The operator first enters an unbalanced list expression. The message

```
|[nnn@`]'|
```

says that (1) the parser recognizes an openning '[', followed by three numerals, indicated by 'nnn'. The suffix @`]' indicates that parsing fails at the character ']'.

Parsing always resumes just beyond the detected syntax error with no attempt at recovery. Next, the operator enters a new-line and then types `[a.b]`. This leads to a sequence of three values. The first,

```
|..[i@‘.’|   ,
```

indicates that parsing failed at a ‘`.`’ after noting an openning ‘`[`’ and one literal (indicated by the ‘`i`’).

Parsing resumes after the period, to produce the literal `B`.

After the ‘`B`’ parsing fails again at the ‘`]`’, now regarded as an unbalanced delimiter because of the intervening syntax error.

In turn, this list is evaluated to yield

```
[ |val/[nnn@‘]’|  NL
  |val/..[i@‘.’|  |ubi:B|  |val/@‘]’|  NL
  ···]
```

The prefix ‘`val/`’ says that an attempt was made to evaluate the parser's errons. The third erron says that the successfully parsed literal `B` is unbound.

Daisy's treatment of errors is exploratory. Though an erron's text is occasionally illuminating, it is often enough just to see where an error arises. Most programmers say that they get their debugging clues from the suffix of an error message. This is particularly true for parsing errors, where the suffix shows the character at which the failure occurs. Here is a quiz: find the syntax error reported in the following Daisy session.

```
&MapVector = \[F V].
&   rec:[ [LOOP HELP]
&         [ \[N ! Ns].  [F:N ! HELP:Ns]
&           \Ns.  if:[ nil?:Ns [] LOOP:Ns ]
&       ]]
 |val/i=\l.i:[l[ff@‘]’| |val/@‘]’|&
&
```

Reading from the right, a ‘`]`’ does not balance an opening ‘`[`’. Here is the

correspondence between the first erron's text and the progress in parsing.

```
  i    ⇒    MapVector
  =    ⇒              =
  \    ⇒                 \
  l    ⇒                  [F V]
  .    ⇒                       .
  i    ⇒        rec
  :    ⇒           :
  [    ⇒            [
  l    ⇒              [LOOP HELP]
  [    ⇒              [
  f    ⇒                \[N ! Ns]. [F:N ! HELP:Ns]
  f    ⇒                \Ns. if:[ nil?:Ns [] LOOP:Ns ]
@‘]’   ⇒               ]
```

2.5 Queries

Diagnostic messages are sometimes issued by the underlying system. Most often seen are messages about host input/output status. In the example below, the operator mistakenly calls for input from a nonexistent file.

```
▮ Daisy -i nonesuch

DSI(002) 11/4/86
query:  [H] Can’t put in.
|scn/dvc/|
&
```

The message

`query:  [H] Can’t put in`

reports the failure to open the `nonesuch` file. This message is output directly to the operator's terminal; it is not part of the interpreter's output. All such messages begin with the text '`query:`' and may be ignored. The input failure is also recorded in the erron `|scn/dvc|`, which reports a scanning error, due the failure of a device handler.

Chapter 3

Daisy operations

An *operation* is a primitive information-processing procedure provided by the Daisy interpreter, its underlying list-processing system, or the implementation host. Operations are grouped according what kinds of arguments they expect and results they produce. The convention used in describing operations gives the expected form of actual arguments and results. Here is an example, describing the `add` operation:

`add` : [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$

> *Addition. Numeral $\mathcal{N}_3$ represents the sum of numbers $\mathcal{N}_1$ and $\mathcal{N}_2$; that is, $\mathcal{N}_3 = \mathcal{N}_1 + \mathcal{N}_2$.*

The first line states that an application of `add` expects the argument to be a list [$\mathcal{N}_1$ $\mathcal{N}_2$] of two numbers, and yields a number, $\mathcal{N}_3$. This is followed by a brief description of `add`. The variables denote references to number objects. The clause "$\mathcal{N}_3 = \mathcal{N}_1 + \mathcal{N}_2$" is intuitive. The `add` operation sums binary fields *in* the objects $\mathcal{N}_1$ and $\mathcal{N}_2$ and places the result in a new object, $\mathcal{N}_3$.

`Add` is a *binary arithmetic* operation. Related operations are `sub`, `mpy`, `div`, `rem`, `and`, `or`, and `xor`. These are grouped in a subsection entitled **Binary Arithmetic**, whose introduction describes what properties they share. For example, none of these operations *requires* an argument of length two. If `add` is given a three-number list, it sums the first two.

3.0.1 Kinds of Values

Distinct variables are used for different kinds of values, according to the table below. These types are usually reiterated in descriptions. The variable

$\mathcal{V}$ is used for values of arbitrary structure or kind.

Types in descriptors		
	$\mathcal{B}$ – `T` *or* `[]`	
$\mathcal{N}$ – a numeral	$\mathcal{L}$ – a list	$\mathcal{C}$ – a character
$\mathcal{I}$ – a literal	$\mathcal{A}$ – an atom	$\mathcal{E}$ – an expression
	$\mathcal{V}$ – a value	

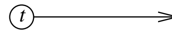
An *atom* can be a directive, a number, or a literal. Nil is also an atom.

A *character* $\mathcal{C}$ is just a literal with a single-character print name. A box (e.g. [A], [NL], etc.) is often used to emphasize characters. The boxes contain mnemonics when the character is not a display code.

A uniform list of indefinate length is indicated by `[` $\mathcal{V}_0 \cdots \mathcal{V}_n$ `]` or `[` $\mathcal{V}_0\ \mathcal{V}_1 \cdots$ `]`. The latter objects are sometimes called *streams*.

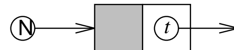
3.0.2 Objects

Daisy's underlying list processing system maintains a heap of uniformly sized cells in three formats. A cell can hold zero, one, or two *citations;* the rest of the cell is binary data. A citation



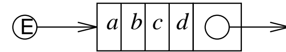
has a one of seven distinct tags and also holds a reference to an object (unless it is a directive; see below). A **nullary cell** looks like

The shading indicates binary data. Daisy does not use this kind of cell [as of release 4]. A *unary cell* looks like

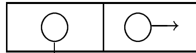


The tag 'N' indicates a unary cell citation. The cited object represents a Daisy numbers. What appears as 42 is a citation to a cell holding the binary representation of the constant 42. A number's tail-citation is a constant, void.

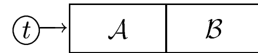
The tag 'E' also indicates a unary cell citation. The cited object represents an *erron,* or error-value, whose binary content is usually a sequence of character codes.



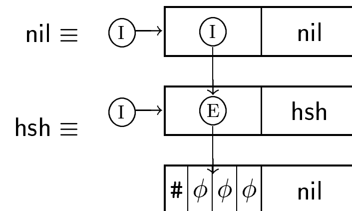
Hence, objects of this kind are depicted as A **binary cell** contains two citations,



and represent Daisy's composite structures. Four tags, 'I' (identifier), 'L' (list), 'A' (application), and 'F' (function), cite binary cells. The cell



represents an identifier object, a list object, an application object, or a function object, depending *only* on the tag t in the citation. There can be differenly tagged citations to the same object.
Nil is an identifier object. It's head, hsh, has the structure of a literal, and thus appears as '#'; but hsh is distinct from the literal #.



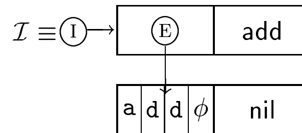
A **directive** is a citation whose content is binary data, and *not* the address of an object. The tag 'D' indicates a directive. indicates that the *content* of the citation is binary data and not the address of another cell. A binary cell *whose head is a directive* looks like this

The constant void in the tail-field of a number is a directive.

3.0.3 Attributes of Operations

There are three entities that can be associated with every operation, a name, an encoding, and an implementation. The encoding is a directive,

used in interpretation to transfer control to the program segment that implements the operation. Operation's names are established by assigning the appropriate directive to a literal identifier. For example, there is a procedure that, given a list of two numbers, creates the number representing their sum. This procedure is invoked by the interpreter when it applies the addition directive, `add`. Should this constant be displayed, it appears as some number with a '`.`' prefix (e.g. `.72`). Initially, `add` is assigned (i.e. globally bound) to the literal `add`. The assignment establishes a structure



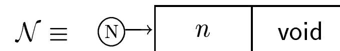
Where $\mathcal{I}$ is the unique instance of `add` in the list space.

3.0.4 Tests

Values $\mathcal{B}$ are the results of *tests*. The notion of a test is that it answers a question about its argument. Tests are used in conditional expressions to select among alternative; however, the `if`-operation accepts *any* value in the position of a test. The value `[]` is interpreted as *false*. Anything else is interpreted as *true*. Hence, any expression can serve for predication. Daisy's test-operations return the literal `T` when their answer is affirmative.

3.0.5 Arithmetic Operations

An *arithmetic operation* tests or manipulates numbers. Each number contains a 32-bit field of binary information, which is displayed as a signed integer using a two's complement interpretation. The variables $\mathcal{N}$ denote cells containing a binary constants,



Host primitives are used to manipulate the field containing n within such cells. There is no provision for arithmetic overflow; the result is a number with a with a truncated binary value.

Arithmetic operations cannot be applied to non-numbers. Operand validation is shared by related operations; for instance, binary operations like `add`, `sub`, `mpy`, and `div` use the same validation routine. A validation error, produces a *non-numeric-argument* erron, but does not report which operation was to be applied.

3.0.6 Reference Operations

Reference operations fall into three categories, although there is some overlap. The first has to do with the logical view of symbolic data in Daisy programming, which distinguishes atoms (literals and numbers) and composite lists.

3.0.7 Character Manipulation

A second set of reference operations is oriented toward text processing. Characters are not a special type, they are simply literals with the property of having a single-letter name. However, auxiliary information is associated with characters and used by the parsing and display operations. This collection of operations gives access to this information.

3.0.8 Object Manipulation

This third set of reference operations directly addresses the underlying representation of Daisy objects. In particular, they provide means to build and explore program representations.

3.0.9 Interface Operations

An *interface* operation applies to the host input-output system, providing a translation between Daisy's logical view of a device as a list of characters and the host's representation as a buffer.

3.0.10 Special Operations

Operations, such as `let`, which invoke evaluation take the implicit environment as a oparand. In this sense, they are part of the Daisy language.

3.1 Arithmetic Tests

Each unary test expects a numeral as an argument; it the argument is not a numeral, the application is erroneous. In particular, the argument cannot be a list: `zero?:<5>` is erroneous and not merely false. The expression should be `zero?:5`.

The binary tests perform comparisons on the numeric fields in two numerals. The comparison operations expect lists whose first two elements are numerals. There is no further validation of argument structure. For example, each of the following expressions returns the same value:

```
eq?:<X Y>   eq?:<X Y *>   eq?:<X Y Z>   eq?:<X Y ! Z>
```

`zero?`:$\mathcal{N} \longrightarrow \mathcal{B}$

Test for Zero. If $\mathcal{N}$ is `0` the result is `true`; otherwise it is `[]`.

`one?`:$\mathcal{N} \longrightarrow \mathcal{B}$

Test for One. If $\mathcal{N}$ is `1` the result is `true`; otherwise it is `[]`.

`neg?`:$\mathcal{N} \longrightarrow \mathcal{B}$

Test for negative. If $\mathcal{N}$ is negative the result is `true`; otherwise it is `[]`.

`pos?`:$\mathcal{N} \longrightarrow \mathcal{B}$

Test for positive. If $\mathcal{N}$ is non-negative the result is `true`; otherwise it is `[]`.

EXAMPLES

$\mathcal{N}$	zero?:$\mathcal{N}$	one?:$\mathcal{N}$	neg?:$\mathcal{N}$	pos?:$\mathcal{N}$
-2	[]	[]	true	[]
-1	[]	[]	true	[]
0	true	[]	[]	true
1	[]	true	[]	true
2	[]	[]	[]	true

`lt?` : [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow$ $\mathcal{B}$

Less-than. If $\mathcal{N}_1$ is less than $\mathcal{N}_2$ the result is `true`; otherwise it is `[]`.

`le?` : [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow$ $\mathcal{B}$

At-most. If $\mathcal{N}_1$ is less than or equal to $\mathcal{N}_2$ the result is `true`; otherwise it is `[]`.

`eq?` : [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow$ $\mathcal{B}$

Numeric equality. If $\mathcal{N}_1$ is numerically equal to $\mathcal{N}_2$ the result is `true`; otherwise it is `[]`. There is also an operation `same?`, which tests either reference or numeric equality.

`ne?` : [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow$ $\mathcal{B}$

Numeric inequality. If $\mathcal{N}_1$ is numerically unequal to $\mathcal{N}_2$ the result is `true`; otherwise it is `[]`; `ne?` inverts the `eq?` test.

`ge?` : [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow$ $\mathcal{B}$

At-least. If $\mathcal{N}_1$ is greater than or equal to $\mathcal{N}_2$ the result is `true`; otherwise it is `[]`; `ge?` inverts the `le?` test.

`gt?` : [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow$ $\mathcal{B}$

Greater-than. If $\mathcal{N}_1$ is greater than $\mathcal{N}_2$ the result is `true`; otherwise it is `[]`; `gt?` inverts the `lt?` test.

EXAMPLES

$\mathcal{V}$	`lt?`: $\mathcal{V}$	`le?`: $\mathcal{V}$	`eq?`: $\mathcal{V}$	`ne?`: $\mathcal{V}$	`ge?`: $\mathcal{V}$	`gt?`: $\mathcal{V}$
`[0 1]`	`true`	`true`	`[]`	`true`	`[]`	`[]`
`[0 0]`	`[]`	`true`	`true`	`[]`	`true`	`[]`
`[1 0]`	`[]`	`[]`	`[]`	`true`	`true`	`true`

3.2 Unary Arithmetic

These operation expects a single numeral; if the argument is not a numeral, the application is erroneous. In particular, the argument cannot be a lists. `inc:[5]` is erroneous; the expression should be `inc?:5`.

`neg` $:\mathcal{N}_1 \longrightarrow \mathcal{N}_2$

> *Negate.* $\mathcal{N}_2$ represents the negation of $\mathcal{N}_1$; that is $\mathcal{N}_2 = -\mathcal{N}_1$.

`sgn` $:\mathcal{N} \longrightarrow$ `-1` or `1`

> *Sign projection.* If $\mathcal{N}$ is negative, the result is `-1`; otherwise it is `1`. In Daisy, `sgn` is `(\N. if:[ neg?:N -1 1 ])`.

`inc` $:\mathcal{N}_1 \longrightarrow \mathcal{N}_2$

> *Increment.* $\mathcal{N}_2$ represents a number one greater $\mathcal{N}_1$; that is $\mathcal{N}_2 = 1 + \mathcal{N}_1$. In Daisy, `inc` is `(\N. add:[1 N])`.

`dcr` $:\mathcal{N}_1 \longrightarrow \mathcal{N}_2$

> *Decrement.* $\mathcal{N}_2$ represents a number one less than $\mathcal{N}_1$; that is $\mathcal{N}_2 = \mathcal{N}_2 - 1$. In Daisy, `dcr` is `(\N. add:[-1 N])`.

`inv` $:\mathcal{N}_1 \longrightarrow \mathcal{N}_2$

> *Invert.* The binary field in $\mathcal{N}_2$ is the bit-wise complement of that in $\mathcal{N}_1$; that is $\mathcal{N}_2 = \overline{\mathcal{N}_1}$.

The second column of examples below reflects the two's complement interpretation of binary fields used in Daisy's display primitives. An inverted binary zero is the two's complement representation of `-1`.

EXAMPLES

$\mathcal{N}$	neg:$\mathcal{N}$	inv:$\mathcal{N}$	inc:$\mathcal{N}$	dcr:$\mathcal{N}$
5	-5	-6	6	4
-3	5	2	-2	-4
0	0	-1	1	-1

3.3 Binary Arithmetic

These operations invoke host arithmetic primitives on the numeric fields in two numerals. The result is a new numeral holding the result. There is no validation of argument structure beyond the first two numerals. For example, the expressions `add:[X Y]`, `add:[X Y *]`, `add:[X Y Z]`, and `add:[X Y ! Z]` each sums the values of `X` and `Y`:

`add`:[$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$

> *Add.* Numeral $\mathcal{N}_3$ represents the sum of numerals $\mathcal{N}_1$ and $\mathcal{N}_2$; that is, $\mathcal{N}_3 = \mathcal{N}_1 + \mathcal{N}_2$.

`sub`:[$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$

> *Subtract.* Numeral $\mathcal{N}_3$ represents the difference of numerals $\mathcal{N}_1$ and $\mathcal{N}_2$; that is, $\mathcal{N}_3 = \mathcal{N}_1 - \mathcal{N}_2$.

`mpy`:[$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$

> *Multiply.* Numeral $\mathcal{N}_3$ represents the product of numerals $\mathcal{N}_1$ and $\mathcal{N}_2$; that is, $\mathcal{N}_3 = \mathcal{N}_1 \cdot \mathcal{N}_2$.

`div`:[$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$

> *Divide.* Numeral $\mathcal{N}_3$ represents the integer quotient of numerals $\mathcal{N}_1$ and $\mathcal{N}_2$; that is, $\mathcal{N}_3 = \mathcal{N}_1 \div \mathcal{N}_2$.
>
> NOTE [March 11, 1996]: Division by zero causes abnormal termination *via* a host exception. This an implementation error and will be corrected.

`rem`:[$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$

> *Remainder.* Numeral $\mathcal{N}_3$ represents the remainder on division of $\mathcal{N}_1$ by $\mathcal{N}_2$. That is, $\mathcal{N}_1 = (\mathcal{N}_1 \div \mathcal{N}_2) \cdot \mathcal{N}_2 + \mathcal{N}_3$.

EXAMPLES

$\mathcal{L}$	`add:`$\mathcal{L}$	`sub:`$\mathcal{L}$	`mpy:`$\mathcal{L}$	`div:`$\mathcal{L}$	`rem:`$\mathcal{L}$
[3 2]	5	1	6	1	1
[-3 5]	2	-8	-15	0	-3
[5 -3]	2	8	-15	-1	2
[0 7]	7	-7	0	0	0
[2 3]	5	-1	6	0	2

Operations `div` and `rem` give the integer quotient and remainder. The function `\M. \N. add:[ mpy:[ M div:[N M] ] rem:[N M] ]` is equivalent to `(\M.\N.M)` provided `M` is not zero.

3.4 Binary "Logic" Operations

The bit-logical operations are faithful to a 1 = *true* interpretation. Thus, 7 would serve as a mask for the three low-order bits of a binary field.

`or` : `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow \mathcal{N}_3$

> *Logical-or.* The numeric field in numeral $\mathcal{N}_3$ contains the bit-wise "logical or" of fields in $\mathcal{N}_1$ and $\mathcal{N}_2$. That is, $\mathcal{N}_3 = \mathcal{N}_1 \oplus \mathcal{N}_2$, where for bits a and b, $a \oplus b = 0$ just when both a and b are 0.

`and` : `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow \mathcal{N}_3$

> *Logical-and.* The numeric field in numeral $\mathcal{N}_3$ contains the bit-wise "logical and" of fields in $\mathcal{N}_1$ and $\mathcal{N}_2$. That is, $\mathcal{N}_3 = \mathcal{N}_1 \odot \mathcal{N}_2$, where for bits a and b, $a \odot b = 1$ just when both a and b are 1.

`xor` : `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow \mathcal{N}_3$

> *Exclusive-or.* The numeric field in numeral $\mathcal{N}_3$ contains the bit-wise "exclusive or" of fields in $\mathcal{N}_1$ and $\mathcal{N}_2$. That is, $\mathcal{N}_3 = \mathcal{N}_1 \otimes \mathcal{N}_2$, where for bits a and b, $a \otimes b = 1$ just when $a \neq b$.

EXAMPLES

$\mathcal{L}$	`and`:$\mathcal{L}$	`or`:$\mathcal{L}$	`xor`:$\mathcal{L}$
[1 1]	1	1	0
[1 0]	0	1	1
[0 1]	0	1	1
[0 0]	0	0	0
[7 5]	5	7	2
[15 5]	5	15	10

3.5 Reference Tests

This collection of reference operations maintains the abstract view of Daisy's symbolic data space of atoms and lists. There is some intersection with the set of *object manipulation* operations described later.

`nil?`: $\mathcal{V} \longrightarrow \mathcal{B}$

> *Test for Nil.* If the argument is `[]` the result is `true`; otherwise `[]`. The `nil?` operation reverses the sense of a test; that is `if:<`$P\ E\ E'$`>` is like `if:< nil?:`$P\ E'\ E$`>` .

`isNML?`: $\mathcal{V} \longrightarrow \mathcal{B}$

> *Test for a numeral.* The result is `true` if the argument is a numeral. See the section on *Tag Tests.*

`isLtrl?`: $\mathcal{V} \longrightarrow \mathcal{B}$

> *Test for a literal.* The result is `true` if the argument is a literal symbol. This means that $\mathcal{V}$ is an identifer object whose head is an erron (serving as a print name).

`isAtm?`: $\mathcal{V} \longrightarrow \mathcal{B}$

> *Test for an Atom.* The result is `true` if the argument is a directive, Nil, a literal, or a numeral. Otherwise, `isAtm?` returns `[]`.

`isLST?`: $\mathcal{V} \longrightarrow \mathcal{B}$

> *Test for a list.* The result is `true` if the argument is a list. See the section on *Tag Tests.*

`same?`: `[`$\mathcal{U}\ \mathcal{V}_1\ \cdots\ \mathcal{V}_n$`]` $\longrightarrow \mathcal{B}$

> *Reference equality.* The result is `true` if object $\mathcal{U}$ is are either truely identical (i.e. the same citation) or numerically equal some $\mathcal{V}_i$. The `same?` operation is normally used as a binary comparison.

EXAMPLES

$\mathcal{V}$	nil?:$\mathcal{V}$	isNML?:$\mathcal{V}$	isLtrl?:$\mathcal{V}$	isAtm?:$\mathcal{V}$	isLST?:$\mathcal{V}$
[]	true	[]	[]	true	[]
17	[]	true	[]	true	[]
red	[]	[]	true	true	[]
[red car]	[]	[]	[]	[]	true

There is a unique literal with any given spelling; so two occurances of a literal symbol are identical in the sense that they display the same reference. Lists do not enjoy this property. Two occurances of an expression such as [E ! E'] produce distinct list objects, which fail a same? test.

Expression	Value	Reason
same?:<"bob" "bob">	true	The literal bob is unique.
same?:<X X>	true	Environments give unique bindings.
same?:<<> <>>	true	Nil is unique.
same?:<5 inc:4>	true	same? tests numeric equality.
same?:<<5> <5>>	[]	List values are not unique.

3.6 List Processing Operations

Most of the operations in this section manipulate list structure. A few are more general but involve related manipulations. The surface syntax of lists reflects their internal representation as binary records with two citations (or "pointer fields"); these are called the list's head and tail.

`head`: $[\,\mathcal{V}_1\ !\ \mathcal{V}_2\,] \longrightarrow \mathcal{V}_1$

> *Head of a list.* If the argument is a list, the result is that list's head.

`tail`: $[\,\mathcal{V}_1\ !\ \mathcal{V}_2\,] \longrightarrow \mathcal{V}_1$

> *Tail of a list.* If the argument is a list, the result is that list's tail.

`cons`: $[\,\mathcal{V}_1\ \ \mathcal{V}_2\,] \longrightarrow [\,\mathcal{V}_1\ !\ \mathcal{V}_2\,]$

> *List constructor.* In Daisy, `cons` is `(\[V V']. [V ! V'])`.

`any?`: $[\,\mathcal{V}_0\ \ \mathcal{V}_1\ \cdots\,] \longrightarrow [\,\mathcal{V}_i\ \ \mathcal{V}_{i+1}\ \cdots\,]$

> *Locate a non-Nil element.* The `any?`-operation returns the greatest suffix of its argument headed by a non-Nil element. In effect,

```
any? = \L.  let:[ [E ! L']
                  L
                  if:] nil?:L  ]]
                       nil?:E  any?:L'
                               L
                      ]
                 ]
```

`all?`: $[\,\mathcal{V}_0\ \ \mathcal{V}_1\ \cdots\,] \longrightarrow \mathcal{B}$

> *Test for no null elements.* The `all?` operation returns `true` if none of the argument's elements is Nil. In Daisy,

```
all? = \L.  let:[ [E ! L']
                   L
                   if:] nil?:L  "T"
                        nil?:E  ]]
                                any?:L'
                       ]
                  ]
```

`in?`:[$\mathcal{U}$ [$\mathcal{V}_0$ $\mathcal{V}_1$ $\cdots$]] $\longrightarrow \mathcal{B}$

Membership Test. The `in?` operation tests whether $\mathcal{V}$ is an element of the list [$\mathcal{V}_0$ $\mathcal{V}_1$ $\cdots$]. In Daisy,

```
in? = \[U L].  let:[ [V ! L']
                     L
                     if:] nil?:L      ]]
                          same?:[U V] "T"
                                      in?:]U L']
                         ]
                    ]
```

`if`:[$\mathcal{T}_0$ $\mathcal{V}_0$ $\mathcal{T}_1$ $\mathcal{V}_1$ $\cdots$] $\longrightarrow \mathcal{V}_i$

Conditional operation. The alternative $\mathcal{V}_i$ corresponding to the first affirmative test value $\mathcal{T}_\rangle$ is returned. Any value other than [] is affirmative. Where the argument has an odd lenght, the last value is returned should all the tests fail. Hence,

`if`:[T_0 E_0 T_1 E_1 E_2] is like `if`:[T_0 E_0 `if`:[T_1 E_1 E_2]]

Given a two-way conditional, `if2`:[$\star$ $\star$ $\star$], Daisy's `if` is specified as

```
if =  \[T V0 ! L] .
        let:[ [ V1 ! L' ]
              L
              if2:] T  V0
                       if2:] nil?:L'  V1
                                      if:L
                   ]        ]
           ]
```

3.6.1 OBSOLETE Remarks

Using the construction functional, one could also specify `in?` as

```
in? = \[U L]. any?:(fc:[same? *]):[ L [U *] ]
```

The most accurate specification is

```
in? = \[E L]. same?:[E ! L]
```

because of the way `same?` extends. In fact, `same?` and `in?` use one implementation and differ only in how they establish their operands.

The `if`-expression is not a special form in Daisy, but has the same intuitive behavior of conditionals. Only tests and the selected alternatives are computed. This behavior is a byproduct of list construction. The following functions define a two-way selector.

```
Predicate = \V. if:] nil?:V  tail  head ]

if2 = \[V1 V2 V3]. (Predicate:V1):[ V2 ! V3 ]
```

The primitive `if` occurs only in the definition of `Predicate` (read: **pred**-ĭ-kayt, *v*), which coerces a value to one of the list access operations `head` or `tail`.

Since `if` is an operation, it can be used in functionals; if `Ps`, `Cs`, and `As` are lists, `[if *]:  [Ps Cs As]` yields a sequence of values selected by the elements in `Ps`.

EXAMPLES

Assume `[N M]` *is bound to* `[0 1]`

Expression	Value
`if:[ nil?:N "A" "B" ]`	B

`if:[ zero?:M  "A"`
`      one?:M  "B"`
`              "C"`
`   ]`
	B

`add:[ if:[ zero?:N  5  7]`
`      if:[ zero?:M  8  9]`
`    ]`
	14

`if:[ if:[ neg?:N`
`             neg?:M`
`             pos?:M`
`         ]`
`     "A"`
`     "B"`
`   ]`
	A

`[if *]:[ [ zero?:N  pos?:N  neg?:N ]`
`         [  "A"      "B"      "C"  ]`
`         [  "D"      "E"      "F"  ]`
`       ]`
	[A B F]

3.7 Object Manipulations

The operations described in this section test and manupulate objects in Daisy's underlying data space. This list processing system maintains a heap of uniformly sized cells in three formats. A cell can hold zero, one, or two *citations;* the rest of the cell is binary data. A citation, ⓣ⟶, has a one of seven distinct tags and also holds a reference to an object (unless it is a directive; see below).

The introduction discusses the underlying representation of objects in Daisy's underlying data space. Briefly, objects are of uniform size in two formats, unary and binary (the nullary format is not used). Unary objects may be cited as numerals or as errons. Binary objects may be cited as indentifiers, lists, applications, or functions. How an object is cited is determined by a tag, located with the reference to the object.

`TagOf`: $\mathcal{V} \longrightarrow \mathcal{N}$

> *Numeric value of a tag.* The binary value of a tag is given in the table below. *These values are subject to change.* Should one want to use the numeric value of a tag (this is not recommended) it is safer to compute it, perhaps assigning the value a name. For instance, the expression `TagOf:^"anything"` gives the value of the tag "identifier".

TAG VALUES [as of version 4]

tag	0	1	2	3	4	5	6*
symbol	`DCT`	`NML`	`FTN`	`IDE`	`LST`	`APL`	`ERR`
citation	Ⓓ⟶	Ⓝ⟶	Ⓕ⟶	Ⓘ⟶	Ⓛ⟶	Ⓐ⟶	Ⓔ⟶

* *Errons cannot be manipulated*

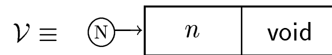
3.8 Tag Tests

`isDCT?`: $\mathcal{V} \longrightarrow \mathcal{B}$

Test for a directive. The result is `true` when $\mathcal{V}$ is a directive.

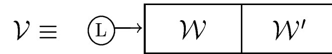
`isNML?`: $\mathcal{V} \longrightarrow \mathcal{B}$

Test for a numeral. If the result is `true`when $\mathcal{V}$ cites a numeral,



`isLST?`: $\mathcal{V} \longrightarrow \mathcal{B}$

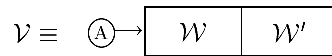
Test for a list-object. The result is `true` when $\mathcal{V}$ cites a binary object with tag L,



List cells represent list expressions and list values in Daisy interpretation.

`isAPL?`: $\mathcal{V} \longrightarrow \mathcal{B}$

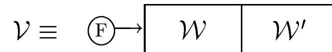
Test for a application-object. The result is `true`when $\mathcal{V}$ cites a binary object with tag A,



Application cells represent application expressions Daisy interpretation.

`isFTN?`: $\mathcal{V} \longrightarrow \mathcal{B}$

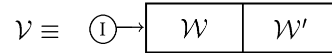
Test for a function-object. The result is `true`when $\mathcal{V}$ cites a binary object with tag F,



Function cells represent function expressions and function closures in Daisy interpretation.

`isIDE?`: $\mathcal{V} \longrightarrow \mathcal{B}$

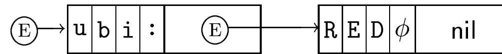
Test for a identifier-object. The result is `true`when $\mathcal{V}$ cites a binary object with tag I,



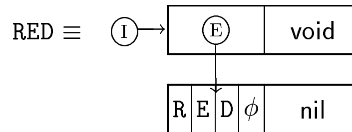
Identifier cells represent literals and quotations in Daisy interpretation.

`isERR?`: $\mathcal{V} \longrightarrow \mathcal{B}$

Test for an identifier-object. If the result is never `true`, but may be `[]`when the argument is not an erron. That is, `isERR?` serves no useful purpose. Manipulation of errons by a Daisy program—even to test for their presence—is by fiat erroneous. Errons are unary cells containing a sequence of characters. For example, the erron `|ubi:RED|` is represented as



The "print name" of a literal is indistiguishible from an erron. The literal `RED` looks like



The `isLtrl?` operation tests for this structure.

3.9 Tag Coercions

Each of the operations below returns a value of the kind indicated if possible. Often, the result is simply a differently tagged citation to the argument. Such coercions are erroneous if they fail to preserve storage classifications; for instance, a list cannot be cited as a numeral.

`asDCT`: $\mathcal{V} \longrightarrow \mathcal{D}$

> *Cite as a directive.* The result $\mathcal{D}$ is a directive, provided $\mathcal{V}$ is a directive or a numeral; it is erroneous to apply `asDCT` to anything else. If $\mathcal{V}$ cites a numeral, $\mathcal{D}$ has its binary content.

`asNML`: $\mathcal{V} \longrightarrow \mathcal{N}$

> *Cite as a numeral.* The result $\mathcal{N}$ is a numeral, provide $\mathcal{V}$ is a numeral or a directive; it is erroneous to apply `asNML` to anything else. If $\mathcal{V}$ is a directive, a numeral is created containing its binary value.

`asIDE`: $\mathcal{V} \longrightarrow \mathcal{I}$

> *Cite as an identifier.* If $\mathcal{V}$ is a binary-object citation, $\mathcal{I}$ is a reference to the *same* object with tag 'I'. Otherwise, application of `asIDE` is erroneous.

`asLST`: $\mathcal{V} \longrightarrow \mathcal{L}$

> *Cite as a list.* If $\mathcal{V}$ is a binary-object citation, $\mathcal{L}$ is a reference to the *same* object with tag 'L'. Otherwise, application of of `asLST` is erroneous.

`asAPL`: $\mathcal{V} \longrightarrow \mathcal{A}$

> *Cite as an application.* If $\mathcal{V}$ is a binary-object citation, $\mathcal{A}$ is a reference to the *same* object with tag 'A'. Otherwise, application of of `asAPL` is erroneous.

`asFTN`: $\mathcal{V} \longrightarrow \mathcal{F}$

> *Cite as a function.* If $\mathcal{V}$ is a binary-object citation, $\mathcal{F}$ is a reference to the *same* object with tag 'F'. Otherwise, application of of `asAPL` is erroneous.

`asERR`: $\mathcal{V} \longrightarrow \mathcal{M}$

> *Error.* The result $\mathcal{M}$ is the erron `|tag/|` unless $\mathcal{V}$ is itself an erron, in which case, $\mathcal{V}$ is returned. See the section titled **Errons**.

3.9.1 Remarks

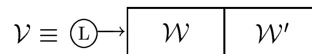
The table below shows what tag coercions are permitted. Conversion between numerals and directives is allowed. Coercions among binary objects are allowed.

	←Coercion→						
	DCT	NML	IDE	LST	APL	FTN	ERR
DCT	*ok*	*ok*	*err*	*err*	*err*	*err*	*err*
NML	*ok*	*ok*	*err*	*err*	*err*	*err*	*err*
IDE	*err*	*err*	*ok*	*ok*	*ok*	*ok*	*err*
LST	*err*	*err*	*ok*	*ok*	*ok*	*ok*	*err*
APL	*err*	*err*	*ok*	*ok*	*ok*	*ok*	*err*
FTN	*err*	*err*	*ok*	*ok*	*ok*	*ok*	*err*
ERR	*err*	*err*	*err*	*err*	*err*	*err*	*err*

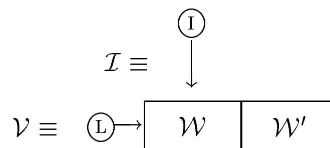
The range of possible directives (24-bit unsigned binary) is smaller than

the range possible numerals (32-bit 2's complement). In `asDCT`, a numeral's high-order bits are truncated to make $\mathcal{D}$. In `asNML`, a directive is 0-extended to make $\mathcal{N}$.

Coercions among identifier, list, application, and function citations do not create new objects. The result is citation with the appropriate tag. For example, suppose the $\mathcal{V}$ cites a list cell



Then the application `asIDE:`$\mathcal{V}$ returns an identifier-citation, $\mathcal{I}$,



and similarly for `asFTN`, `asLST`, and `asAPL`. A binary citation's tag determines how the cited object is displayed.

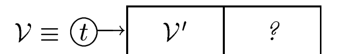
Expression	Value
`asLST:["A" "B"]`	`[A B]`
`asAPL:["A" "B"]`	`A:B`
`asFTN:["A" "B"]`	`\A.B`
`asIDE:["A" "B"]`	`|?I|`

In the fourth example, the erron `|?I|` is generated by the `issue` operation because no syntax is associated with identifiers having this structure.

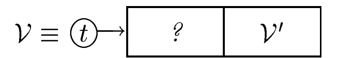
3.10 Access to Composite Objects

The head and tail operations are restricted to list objects. They include a test on the tag of their argument. Two less sensitive versions omit this test. These can be used to access constituents of non-list cells.

_hd : $\mathcal{V} \longrightarrow \mathcal{V}'$

> *Head of a composite object.* The _hd operation returns the citation $\mathcal{V}'$ whenever $\mathcal{V}$ has the form
>
> 
>
> Since tag-insensitive access operations are used to retrieve bindings from environments, _hd is (\[H ! T]. H) but could also be implemented as (\V. head:asLST:V).

_tl : $\mathcal{V} \longrightarrow \mathcal{V}'$

> *Tail of a composite object.* The _tl operation returns the citation $\mathcal{V}'$ whenever $\mathcal{V}$ has the form
>
> 
>
> Since tag-insensitive access operations are used to retrieve bindings from environments, _tl is (\[H ! T]. T) and is approximately implemented as (\V. tail:asLST:V).

3.10.1 Remarks

The operations head and tail do ordinary list access and include a validation that the argument is a list. Since they omit any such test, _hd and _tl are slightly faster. In addition, _tl retreives the tail of *any* object that has one, including numerals and errons.

One can use tag coercion to get at the components of non-list objects. The coercions also document intent. The following program fragment retreives the formal argument of a function object.

```
FormalArgument = \F.
  if:[ isFTN?:F
          head:asLST:F
          "mistaken function-access"
      ]
```

In this definition, an explicit test confirm that `F` is a function cell. At the same time, there is a lot of redundant tag checking by `head` and `asLST`. Since it is assured that `F` is a binary object, one might as well write

```
FormalArgument = \F.      ;
  if:[ isFTN?:F           ; Hence, F is binary
          _hd:F           ;
          "mistaken function-access"
      ]                   ;
```

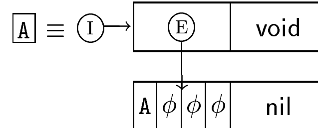
NOTE[March 11, 1996]: Use of `_hd` and `_tl` in environment look-up leads to some baffling Daisy bugs. For example, the expression

`(\[H!T].T):5`

returns the void directive because numerals have tail-citations. Even more inscrutable are assignment effects, through which global function bindings end up in environments. More sensitive binding retrieval is under consideration.

3.11 Characters

Daisy characters are simply those literals with a single-character display names. The character `A` is the literal cell



`Chr?`: $\mathcal{V} \longrightarrow \mathcal{B}$

> *Character test.* The `Chr?` operation returns `true` if its argument is single-character literal.

`ChrAsNml`: $\mathcal{C} \longrightarrow \mathcal{N}$

> *Character's numeric code.* If the argument is a character, `ChrAsNml` builds a numeral, $\mathcal{N}$, holding the binary value of its display code.

`NmlAsChr`: $\mathcal{N} \longrightarrow \mathcal{C}$

> *Convert a numeral to a character.* If the argument is a numeral, $\mathcal{C}$ cites the character whose display code is equal to the seven low-order bits of $\mathcal{N}$'s binary content.

EXAMPLES

Expression	Value
`Chr?:[]`	`[]`
`Chr?:"RED"`	`[]`
`Chr?:"R"`	`true`
`Chr?:5`	`[]`
`Chr?:"5"`	`true`
`ChrAsNml:"5"`	`53`
`NmlAsChr:82`	`R`
`NmlAsChr:338`	`R`

3.12 Character Classification

Parsing, scanning, and display operations employ an internal classification of host character codes. Though this classification cannot be altered by Daisy programs, tests against it are permitted. The classifications are explained in descriptions of `scan` and `parse` and summarized in Figures ? and ?. There is no test operations for *Scanning Escape* (the back-quote character `` ` ``), *Literal quote* (the double-quote character `"`), *Comment* (the bar character `|`), or *Sign* (the characters `+` and `-`).

`ScnSPC?`: $\mathcal{C} \longrightarrow \mathcal{B}$

> *Space-character test.* The result is `true` if $\mathcal{C}$ is classified as a *space* character (SPC in Figure ?; see also `scan`). The spaces are `␣` and `HT`.

`ScnDGT?`: $\mathcal{C} \longrightarrow \mathcal{B}$

> *Digit-character test.* The result is `true` if character $\mathcal{C}$'s is a *digit* (DGT in Figure ?; see also `scan`). The digits are `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, and `9`.

`ScnLFA?`: $\mathcal{C} \longrightarrow \mathcal{B}$

> *Alpha-character test.* The result is `true` if character $\mathcal{C}$ is *alphabetic* (LFA in Figure ?; see also `scan`). The alphabetic characters are `A` through `Z` and `a` through `z`.

`ScnNON?`: $\mathcal{C} \longrightarrow \mathcal{B}$

> *Neutral-character test.* The result is `true` if character $\mathcal{C}$ is classified as *neutral* (NON in Figure ?; see also `scan`). The neutral characters are `#`, `%`, `&`, `'`, `,`, `/`, `;`, `@`, `_`, and `~`.

`ScnSYM?`: $\mathcal{C} \longrightarrow \mathcal{B}$

> *Symbol-character test.* The result is `true` if character $\mathcal{C}$ is classified as a *symbol* (SYM in Figure ?; see also `scan`). Symbols have an auxiliary classification, of meaning in parsing. The symbols are control characters `NL`, `VT`, `FF`, and `CR`; and display characters `!`, `$`, `(`, `)`, `*`, `.`, `:`, `5`, `<`, `=`, `>`, `[`, `\`, `]`, `^`, `{`, and `}`.

`ScnCTL?`: $\mathcal{C} \longrightarrow \mathcal{B}$

Control-character test. The result is `true` if character $\mathcal{C}$ is classified as *control* (CTL in Figure ?; see also `scan`). The control codes are

NUL SOH STX ETX ENQ ACK BEL BS SO SI DLE DC1 DC2

DC3 DC4 NAK SYN ETB CAN EM SUB ESC FS GS RS US DEL

3.13 Sequencers

Certain programs require a degree of control over the order of evaluation. In particular, programs that manage external events (input and output) may need to defer using a list until one of its field is known to be present. The sequencing operations provide a crude form of control over order of evaluation.

`crc_hd`: $\mathcal{V} \longrightarrow \mathcal{V}$

> *Coerce-head.* The argument, normally a list, is returned once its head is present.
>
> ```
> crc_hd = \L.  let:[ [H ! T]  L  if:[ H  L  L ] ]
> ```

`crc_tl`: $\mathcal{V} \longrightarrow \mathcal{V}$

> *Coerce-tail.* The argument, normally a list, is returned once its tail is present.
>
> ```
> crc_tl = \L.  let:[ [H ! T]  L  if:[ T  L  L ] ]
> ```

`seq`: [$\mathcal{V}_0$ $\mathcal{V}_1$ $\cdots$ $\mathcal{V}_n$] $\longrightarrow \mathcal{V}_n$

> *Sequencer.* The last element of the list argument is returned with all preceding elements having been made present. In Daisy,
>
> ```
> seq = ^\L. let:[ [H ! T]  crc_hd:L  if:[ nil?:T  H  seq:T ] ]
> ```

3.13.1 Remarks

The `crc_hd` and `crc_tl` operations act as identity functions, but access the head and tail of their arguments before returning them. The expressions `same?:[X crc_"hd:X]` and `same?:[X crc_"tl:X]` always return `true`where the accessed fields are defined. That is, if Ω is a divergent expression, then <Ω E> has a value but `crc_hd:<`Ω E`>` does not, and similarly for `crc_tl`.

Since the `if` operation uses its first argument, it can be used to get the effect of `crchd` and `crctl`, as illustrated in the descriptions above. An alternative specification exploits the fact application forces the argument to be present before invoking a function:

```
crc_hd = ^\L. (\X.L):_hd:L
crc_tl = ^\L. (\X.L):_tl:L
```

The following program has been used as a tracing facility:

```
Trace = ^\Message. \Value.
   seq:[ screen:issue:Message
         Value
        ]
```

The value of `(Trace:["X is " X])` acts as an identity operation but displays `[X is` *value*`]` on the terminal when it is applied. The sequencers are needed to address certain problems in event coordination, but this is not one of them. `Trace`, whose only purpose is an (output) effect, is not an applicative construct. On the other hand, instrumentation of applicative programs is subject of current research. See, for example, "*Debugging in Applicative Languages,*" by O'Donnell and Hall (Indiana University Computer Science Department Technical Report No. 223, 1987).

A more stylistic example is the problem of merging two streams of data in a timely fashion, where the requirement is that an item in a stream exists before it is selected. The `Merge` defined below is not timely because it does not guarantee timeliness.

```
Merge = ^\[As Bs].
  head:{ [ head:As ! Merge:[Bs tail:As] ]
         [ head:Bs ! Merge:[As tail:Bs] ]
       }
```

The *list* `[ head:As !  Merge:[Bs tail:As] ]` can exist before `head:As` produces a value. To assure the presence of an item before committing its use, one could instead define

```
Merge = ^\[As Bs].
  head:{ crc_hd:[ head:As ! Merge:[Bs tail:As] ]
         crc_hd:[ head:Bs ! Merge:[As tail:Bs] ]
       }
```

3.14 Interface Operations

Input to and output from Daisy are done by host-interface operations, that produce and consume lists of characters. The input operations are `console` and `dski`; output is done by `screen` and `dsko`. `console` and `screen` effect the interactive operator's terminal; `dski` and `dsko` effect the host file system.

`console`:$\mathcal{I} \longrightarrow$ `[`$\mathcal{C}_0\ \mathcal{C}_1\ \cdots$`]`

> *Interactive input.* The literal $\mathcal{I}$ is displayed as a prompt for input from the operator's keyboard (`stdin` in UNIX). The result is the list of characters corresponding to key strokes. Several instances of `console` can be in effect. An input channel is terminated by typing EOT (control 'D'). In the case that EOT is the *only* thing typed the list that results is `[`EOT`]` .

`screen`:`[`$\mathcal{C}_0\ \mathcal{C}_1\ \cdots$`]` $\longrightarrow$ `[]`

> *Interactive output.* The characters in the argument are displayed on the operator's terminal screen (`stdout` in UNIX). Nil is returned *after* the entire argument has been displayed. That is, `screen` does not terminate until it has consumed all of the elements of the character stream.

`dski`:$\mathcal{I} \longrightarrow$ `[`$\mathcal{C}_0\ \mathcal{C}_1\ \cdots$`]`

> *File input.* The literal $\mathcal{I}$ names a host text file. The result is the list of characters read from that file.

`dsko`:`[`$\mathcal{I}$`[`$\mathcal{C}_0\ \mathcal{C}_1\ \cdots$`]]` $\longrightarrow$ `[]`

> *File output.* The literal $\mathcal{I}$ names a host text file. This list of characters, `[`$\mathcal{C}_0\ \mathcal{C}_1\ \cdots$`]`, is written to that file. `Dsko` returns Nil *after* the entire list has been written.

3.14.1 Remarks

Abstractly, peripheral I/O is buffered directly in Daisy's list-space. In implementation, the host-interface operations copy from host buffers into lists. Synchronization with host buffers depends, in part, how the host processes peripheral I/O. With terminals in normal mode, the host withholds input in order to process line-editing characters (back-space, line-delete, etc.). Hence, key board input is not presented to Daisy until a new-line

is processed. Even in "raw" modes, host I/O is synchronous by default. Asynchronous version of the interface operations are in development.
The host text file

```
Looking for that
    fatal error?
Read your listing
    on the stair.
```

is represented in Daisy as the list

[L o o k i n g ␣ f o r ␣ t h a t NL
HT f a t a l ␣ e r r o r ? NL
R e a d ␣ y o u r ␣ l i s t i n g NL
HT o n ␣ t h e ␣ s t a i r .]

In some cases the EOT character occurs at the end of such a list. A program that echo's terminal input could be written as follows. Daisy output is shaded; operator input is not. The ampersand is Daisy's top-level prompt.

```
&Parrot = ^\P.screen:console:P
 Parrot
&Parrot:"??"
 ??abc
abc
??def
def
??EOT[]
&
```

The list produced by `console` is

[a b c NL d e f NL]

which includes the two carriage-returns entered by the operator, An instance of `console` terminates its list when it encounters the EOT (control-'D') code form the key board.
`Screen`
displays this list. The occurance of prompts ?? reflects the host's buffering of input through new-line entries. The prompts are raised as `screen`

catches up with `console`. Nil, the value of the expression `Parrot:"??"`, is displayed once output is exhausted.

A program that records terminal input on a file `F` could be written as follows.

```
&Tap = ^\F. dsko:[F console:F]
 Tap
&Tap:"IN"  INLooking for that
IN   [HT]  fatal error?
INRead your listing
IN   [HT]  on the stair.
IN [EOT] []
&
```

`Tap` records console input on a file, using the file's name as a prompt. Since `dsko` does not terminate until it has consumed its argument, `Tap` retains control until the `F`-channel to the keyboard is terminated. The host file '`IN`' records console input, which may subsequently be read using `dski`:

```
&dski:"IN"  [L o o k i n g ␣f o r ␣t h a t
   [HT]  f a t a l ␣e r r o r ?
 R e a d ␣y o u r ␣l i s t i n g
   [HT]  o n ␣t h e ␣s t a i r .]
&
```

`Dski`

returns a list of characters corresponding to what was written to the file. The effect of the `screen` operation is a display of these characters

```
&screen:dski:"IN"
 Looking for that
   [HT]  fatal error?
 Read your listing
   [HT]  on the stair.[]
&
```

The structures returned by `console` and `dski` are indistiguishible from other values. They can be manipulated by Daisy programs, including those primitives for parsing and scanning.

3.15 Text Generation

`issue`:$\mathcal{V} \longrightarrow$ `[` $\mathcal{C}_0$ $\mathcal{C}_1$ $\cdots$ `]`

> *Generate text.* The `issue` operation produces a character stream spelling the value $\mathcal{V}$. The result can be used by interface operations. There are some structures that `issue` cannot spell. the sequence '`|I?|`' is issued for identifier cells having no print name. The prefix '`\=?=`' is appended to function closures.

NOTE [March 11, 1996]: revisions to `issue` are in progress, bringing it closer to the `xparse` and `scan` operations.

$\mathcal{V}$	`issue`:$\mathcal{V}$
`BLUE`	`[` B L U E `]`
`[51 X]`	`[` [5 1 ␣ X] `]`
`\A.<A>`	`[` \ A . < A > `]`

MORE TO COME

3.16 Scanning

Scanning is the translation of a sequence of characters, such as would be produced by an interface operation, into atomic symbols. *In this section* the term *text* refers to a list

$$[\,\mathcal{C}_0\ \ \mathcal{C}_1\ \cdots\,]\quad .$$

It is easiest to think of each $\mathcal{C}_i$ as a character, although the scanning operations accept any object in these places.

`scan`: $[\,\mathcal{C}_0\ \ \mathcal{C}_1\ \cdots\,] \longrightarrow [\,\mathcal{A}\ \ \mathcal{C}_j\ \ \mathcal{C}_{j+1}\ \cdots\,]$

> *Scan text.* The `scan` operation incorporates the prefix of text into the atomic object $\mathcal{A}$ and appends it to the suffix beyond $\mathcal{A}$'s spelling. Non-characters are accepted as though they were symbols. That is, if $\mathcal{V}$ is not a character, then `scan`: $[\mathcal{V}\ \ \mathcal{C}_1\ \ \mathcal{C}_2\ \cdots\,]$ returns $[\mathcal{V}\ \ \mathcal{C}_1\ \ \mathcal{C}_2\ \cdots\,]$. The `scan` operation also develops literal quotations. It is erroneous if `scan`'s argument is Nil. Other boundary conditions arise when the argument contains nothing to incorporate; the details are discussed in the remarks below.

`scans`: $[\,\mathcal{C}_0\ \ \mathcal{C}_1\ \cdots\,] \longrightarrow [\,\mathcal{A}_0\ \ \mathcal{A}_1\ \cdots\,]$

`scan`

> *iterated.* The `scans` operation applies `scan` to successive suffixes, producing a sequence of atoms from a sequence of characters. Here is an approximate specification:

```
scans = ^\Characters .

          let:[ [Atom ! Suffix]  scan:Characters

          let:[ Atoms  if:< nil?:Suffix  <>  scans:Suffix >

                < Atom ! Atoms >

               ]]
```

3.16.1 Remarks – Scanning Details

Scanning is done according to a classification of characters described below. Briefly,

- When `scan` encounters a digit or sign it builds a numeral.

- When `scan` encounters a neutral character, such as a letter, it builds a literal.

`Scan`

- it accepts as symbols those characters that have meaning in parsing.

`Scan`

- accepts as a symbol any object other than a character.

- • When `scan` encounters the delimiter ", it builds a literal quotation.

Figure ? gives the character classification with respect to scanning. The classifications are

Class	Meaning	Remark
END	end of text	EOT
CMT	comment	|
SIC	escape	‘
SPC	space	␣, HT
SGN	sign	+ and -
SYM	symbols	e.g. [, :
DGT	digit	0 through 9
LFA	alphabetic	A ··· Z and a ··· z
NON	annotation	neutral display codes, e.g. %
CTL	contol	neutral contol codes

The operations, `ScnCTL?`, `ScnSPC?`, `ScnDGT?`, `ScnLFA?`, `ScnNON?`, and

`ScnSYM?` test characters according to this classification.

Spaces

`scan:[` *space* `!` $\mathcal{L}$ `]` $\longrightarrow$ `scan:`$\mathcal{L}$

Blanks and tabs are skipped. “Vertical” spaces (e.g. NL and FF) are classified as symbols. `scan:[`␣ ··· ␣`]` returns the value `[#]`. The ‘`#`’ is Nil’s head, which `scan` sees as a symbol.

Comments

scan:[| ··· NL ! $\mathcal{L}$] ⟶ [NL ! $\mathcal{L}$]

When it encounters the comment delimiter, |, scan advances to the next new-line and returns it as a symbol.

End-of-Text

*quad*scan:[$\mathcal{C}_0$ ··· $\mathcal{C}_n$ EOT ! $\mathcal{L}$] $\xrightarrow{\approx}$ scan:[$\mathcal{C}_0$ ··· $\mathcal{C}_n$]
scan:[spc ··· spc EOT ! $\mathcal{L}$] ⟶ [NL]

When scan encounters EOT, it is as though the list were Nil at that point. Scanning tests for EOT only after the spelling of an atom; if the prefix leading to EOT contains nothing but spaces and comments, scan introduces a new-line symbol.

Normally, EOT characters do not occur in text. The Daisy input operations, dvci and console, usually terminate text with Nil, although they introduce an EOT in certain pathalogical conditions. Also, the host's escape mechanism can be used to force an EOT into input text.

Symbols

scan:[*symbol* ! $\mathcal{L}$] ⟶ [*symbol* ! $\mathcal{L}$] Characters classified as symbols have meaning in parsing and are accepted as literals. For example,

scans:< "[" "r" "e" "d" "!" "c" "a" "b" "]" ···>

yields [[red ! cab] ···].

Numerals

scan:[*digit* ··· *digit* ! $\mathcal{L}$] ⟶ [$\mathcal{N}$! $\mathcal{L}$]
scan:[+ *digit* ··· *digit* ! $\mathcal{L}$] ⟶ [$\mathcal{N}$! $\mathcal{L}$]
scan:[- *digit* ··· *digit* ! $\mathcal{L}$] ⟶ [$\mathcal{N}$! $\mathcal{L}$] When it encounters prefix of digits, possibly with a leading sign character, scan builds a numeral, $\mathcal{N}$ representing the decimal integer expressed by the digits. For example,

scan:[1 5 8 ␣ A ···]

yields [158 ␣ A ···]. The first three characters are incorporated as a numeral 158.

Literals

scan:[*letter* $\cdots$ *nonletter* ! $\mathcal{L}$] $\longrightarrow$ [$\mathcal{I}$ *nonletter* ! $\mathcal{L}$]

When the leading character is neutral, the prefix of text is incorporated as a literal, $\mathcal{I}$, whose print-name is that prefix. For example,

scan:[R E D ␣ C A R $\cdots$]

yields [RED ␣ c a r $\cdots$]. The first three characters are incorporated as a literal, RED.

The leading character can be an alphabetic character or an annotation character; the signs + and - are accepted when they are not followed by digits. The literal's spelling stops with a space, a comment, EOT, punctuation, or parsing symbol.

The back-quote ‘ serves as an escape in literal formation. Any following character is treated as neutral. For instance,

scan:[A ‘ * ␣ ! $\mathcal{L}$]

produces a list whose head is the literal spelled ‘A*’.

Literal quotation

scan:[" $\mathcal{C}_0$ $\cdots$ $\mathcal{C}_n$ " ! $\mathcal{L}$] $\longrightarrow$ [$\mathcal{I}$! $\mathcal{L}$] Any prefix of text surrounded by double-quotes " is incorporated as a literal quotation. " and ‘ must be preceded by an escape. For instance,

scan:[" ‘ " S a y , ␣ ‘ ‘ X . ’ ‘ " " ! $\mathcal{L}$]

produces *a quotation of* the literal spelled "Say, ‘X.’" (including the double quotes).

Non-character Text

scan:[$\mathcal{V}$! $\mathcal{L}$] $\longrightarrow$ [$\mathcal{V}$! $\mathcal{L}$] A non-character object in scan's argument is treated as a symbol. That is, non-characters are accepted as they are. However, such objects are ignored when they occur between quotation delimiters.

3.17 Parsing

Parsing is the translation of a sequence of *tokens,* such as would be produced by the `scans` operation, into an internally represented Daisy expression. It is easiest to think of a token as an atom (a literal or numeral), but $\mathcal{T}_i$ can be anything. The parsing operations accept non-atomic objects as neutral literals.

`parse`: `[` $\mathcal{C}_0$ $\mathcal{C}_1$ $\cdots$ `]` $\longrightarrow$ `[` $\mathcal{E}_0$ $\mathcal{E}_1$ $\cdots$ `]`

`xparses`

> ∘ `scans`. The `parse` operations is `(\Ts. xparses:scans:Ts)`. It is retained from previous releases of Daisy. In future releases, this will be the `xparse` operation.

`xparse`: `[` $\mathcal{T}_0$ $\mathcal{T}_1$ $\cdots$ `]` $\longrightarrow$ `[` $\mathcal{E}$ $\mathcal{T}_j$ $\mathcal{T}_{j+1}$ $\cdots$ `]`

> *Parse text.* The `parse` operation incorporates the prefix of text into an expression object $\mathcal{E}$. The result is a list whose head is $\mathcal{E}$ and whose tail is the suffix of text beyond $\mathcal{E}$'s spelling. For example,
>
> ```
> parse:< "\" "n" "." "G1" ":" "n" "A" "B" "C" >
> ```
>
> returns the list `[ \n.G1:n  A B C]`, whose first element is a function expression. It is erroneous if `parse`'s argument is Nil.

`xparses`: `[` $\mathcal{T}_0$ $\mathcal{T}_1$ $\cdots$ `]` $\longrightarrow$ `[` $\mathcal{E}_0$ $\mathcal{E}_1$ $\cdots$ `]`

`parse`

> *iterated.* The `xparses` operation applies `xparse` to successive suffixes, producing a sequence of expressions from the sequence of tokens. Here is a partial specification in Daisy.
>
> ```
> xparses = ^\Tokens .
>
>    let:[ [Exp ! Suffix]
>          xparse:Tokens
>    let:[ Exps
>          if:< nil?:Suffix  <>  xparses:Suffix >
>
>          < Expression ! Expressions >
>
>         ]]
> ```

3.17.1 Remarks—Parsing Details

If the argument contains nothing to parse, `parse` returns the list `[#]`, where the `#` displays Nil's head. An example of such an argument is `[` NL ⋯ NL `]`.

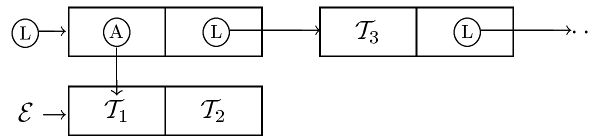
The tokens of interest in parsing are those single-letter symbols of significance in Daisy's surface syntax.

Symbols	Purpose
NL VT FF CR ( )	precedence
^	Value quotation
\ .	Function expression
:	Application expression
< >	List expression
[ ]	List expression
{ }	List expression
!	Dot notation
*	Cyclic tail
=	Assignment expression
$	Strictness annotation

Figure ? shows the correspondence between Daisy's surface language expression representations. The function of parsing is to create expression objects.

$$\texttt{parse:[}\ \mathcal{T}_0\ \boxed{:}\ \mathcal{T}_2\ \mathcal{T}_3\ \cdots\texttt{]}$$

yields a list `[`$\mathcal{E}\ \mathcal{T}_3\ \cdots$`]` whose head, $\mathcal{E}$, cites the application of $\mathcal{T}_1$ to $\mathcal{T}_2$. That is, the result looks like



Parsing is by recursive descent, with single-token look-ahead for infix symbols such as :. Look-ahead is inhibited by occurances of vertical carriage control characters such as NL. This is done for the sake of interaction.

$$\texttt{parse:[inc}\ \boxed{:}\ \boxed{\text{NL}}\ \texttt{255}\cdots\texttt{]}$$

develops an application of `inc` to `255`, so the result appears as `[inc:255` NL `···]`, while the expression

`parse:[ inc` NL `:` `255···]`

stops looking for a colon at the new-line; and thus returns

`[ inc` NL `:` `···]`

In other words, the second `parse` application accepts the literal `inc` as the expression $\mathcal{E}$.

MORE TO COME

3.18 Special Operations

The operations in this section must be regarded as special forms because their invokation involves manipulation of the implicit environment object.

`val`: $\mathcal{E} \longrightarrow \mathcal{V}$

Evaluate.

The result is the value of expression $\mathcal{E}$ using the environment in effect at the point of invokation.

MORE TO COME

`evlst`: `[` $\mathcal{E}_0$ $\mathcal{E}_1$ $\cdots$ `]` $\longrightarrow$ `[` $\mathcal{V}_0$ $\mathcal{V}_1$ $\cdots$ `]`

`val`

iterated..

In Daisy,

```
evlst = \L.
         let:[ [H ! T]
               L
               if:< nil?:L
                       <>
                    same?:<T L>
                       <val:H *>
                       <val:H ! evlst:T>
                  >
             ]
```

MORE TO COME

`let`: `[` $\mathcal{X}$ $\mathcal{E}_1$ $\mathcal{E}_2$ `]` $\longrightarrow \mathcal{V}$

Lexical binder.

MORE TO COME

`rec`: `[` $\mathcal{X}$ $\mathcal{E}_1$ $\mathcal{E}_2$ `]` $\longrightarrow \mathcal{V}$

Recursive binder.

MORE TO COME

`fix` : [$\mathcal{X}$ $\mathcal{E}$] $\longrightarrow \mathcal{V}$

Recursive binder.

MORE TO COME

[

]REFERENCE (a) – Daisy Operations

Daisy Operations

(by group)

Types in descriptors		
	$\mathcal{B}$ – `T` *or* `[]`	
$\mathcal{N}$ – a numeral	$\mathcal{L}$ – a list	$\mathcal{C}$ – a character
$\mathcal{I}$ – a literal	$\mathcal{A}$ – an atom	$\mathcal{E}$ – an expression
	$\mathcal{V}$ – a value	

Arithmetic Tests `zero?`: $\mathcal{N} \longrightarrow \mathcal{B}$ . . . . *Test for Zero*

`one?`: $\mathcal{N} \longrightarrow \mathcal{B}$ . . . . *Test for One*

`neg?`: $\mathcal{N} \longrightarrow \mathcal{B}$ . . . . *Test for negative*

`pos?`: $\mathcal{N} \longrightarrow \mathcal{B}$ . . . . *Test for positive*

`lt?`: `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow$ $\mathcal{B}$ . . . . *Less-than*

`le?`: `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow$ $\mathcal{B}$ . . . . *At-most*

`eq?`: `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow$ $\mathcal{B}$ . . . . *Numeric equality*

`ne?`: `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow$ $\mathcal{B}$ . . . . *Numeric inequality*

`ge?`: `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow$ $\mathcal{B}$ . . . . *At-least*

`gt?`: `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow$ $\mathcal{B}$ . . . . *Greater-than*

Unary Arithmetic `neg`: $\mathcal{N}_1 \longrightarrow \mathcal{N}_2$ . . . . *Negate*

`sgn`: $\mathcal{N} \longrightarrow$ `-1` or `1` . . . . *Sign projection*

`inc`: $\mathcal{N}_1 \longrightarrow \mathcal{N}_2$ . . . . *Increment*

`dcr`: $\mathcal{N}_1 \longrightarrow \mathcal{N}_2$ . . . . *Decrement*

`inv`: $\mathcal{N}_1 \longrightarrow \mathcal{N}_2$ . . . . *Invert*

Binary Arithmetic `add`: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Add*

`sub`: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Subtract*
`mpy`: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Multiply*
`div`: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Divide*
`rem`: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Remainder*

Binary Logic `or`: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Logical-or*

`and`: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Logical-and*
`xor`: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Exclusive-or*

Reference Tests `nil?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for Nil*

`isNML?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a numeral*
`isLtrl?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a literal*
`isAtm?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for an Atom*
`isLST?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a list*
`same?`: [$\mathcal{U}$ $\mathcal{V}_1$ $\cdots$ $\mathcal{V}_n$] $\longrightarrow \mathcal{B}$ . . . . *Reference equality*

List Processing `head`: [$\mathcal{V}_1$! $\mathcal{V}_2$] $\longrightarrow \mathcal{V}_1$ . . . . *Head of a list*

`tail`: [$\mathcal{V}_1$! $\mathcal{V}_2$] $\longrightarrow \mathcal{V}_1$ . . . . *Tail of a list*
`cons`: [$\mathcal{V}_1$ $\mathcal{V}_2$] $\longrightarrow$ [$\mathcal{V}_1$! $\mathcal{V}_2$] . . . . *List constructor*
`frons`: [$\mathcal{V}$ $\mathcal{V}'$] $\longrightarrow \mathcal{L}$ . . . . *Multiset constructor*
`any?`: [$\mathcal{V}_0$ $\mathcal{V}_1$ $\cdots$] $\longrightarrow$ [$\mathcal{V}_i$ $\mathcal{V}_{i+1}$ $\cdots$] . *Locate a non-Nil element*
`all?`: [$\mathcal{V}_0$ $\mathcal{V}_1$ $\cdots$] $\longrightarrow \mathcal{B}$ . . . . *Test for no null elements*
`in?`: [$\mathcal{U}$ [$\mathcal{V}_0$ $\mathcal{V}_1$ $\cdots$]] $\longrightarrow \mathcal{B}$ . . . . *Membership Test*
`if`: [$\mathcal{T}_0$ $\mathcal{V}_0$ $\mathcal{T}_1$ $\mathcal{V}_1$ $\cdots$] $\longrightarrow \mathcal{V}_i$ . . . . . *Conditional operation*

Object Manipulation `TagOf`: $\mathcal{V} \longrightarrow \mathcal{N}$ . . . . *Numeric value of a tag*

Tag Tests `isDCT?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a directive*

`isNML?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a numeral*

`isLST?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a list-object*

`isAPL?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a application-object*

`isFTN?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a function-object*

`isIDE?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a identifier-object*

`isERR?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for an identifier-object*

Tag Coercion `asDCT`: $\mathcal{V} \longrightarrow \mathcal{D}$ . . . . *Cite as a directive*

`asNML`: $\mathcal{V} \longrightarrow \mathcal{N}$ . . . . *Cite as a numeral*

`asIDE`: $\mathcal{V} \longrightarrow \mathcal{I}$ . . . . *Cite as an identifier*

`asLST`: $\mathcal{V} \longrightarrow \mathcal{L}$ . . . . *Cite as a list*

`asAPL`: $\mathcal{V} \longrightarrow \mathcal{A}$ . . . . *Cite as an application*

`asFTN`: $\mathcal{V} \longrightarrow \mathcal{F}$ . . . . *Cite as a function*

`asERR`: $\mathcal{V} \longrightarrow \mathcal{M}$ . . . . *Error*

Access to Composite Objects `_hd`: $\mathcal{V} \longrightarrow \mathcal{V}'$ . . . . *Head of a compos*

`_tl`: $\mathcal{V} \longrightarrow \mathcal{V}'$ . . . . *Tail of a composite object*

Character Manipulation `Chr?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Character*

`ChrAsNml`: $\mathcal{C} \longrightarrow \mathcal{N}$ . . . . *Character's numeric code*

`NmlAsChr`: $\mathcal{N} \longrightarrow \mathcal{C}$ . . . . *Convert a numeral to a character*

Character Classification `ScnSPC?`: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Space-character t*

`ScnDGT?`: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Digit-character test*

`ScnLFA?`: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Alpha-character test*

`ScnNON?`: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Neutral-character test*

`ScnSYM?`: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Symbol-character test*

`ScnCTL?`: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Control-character test*

Sequencing `crc_hd`: $\mathcal{V} \longrightarrow \mathcal{V}$ . . . . *Coerce-head*

`crc_tl`: $\mathcal{V} \longrightarrow \mathcal{V}$ . . . . *Coerce-tail*
`seq`: $[\,\mathcal{V}_0\ \mathcal{V}_1\ \cdots\ \mathcal{V}_n\,] \longrightarrow \mathcal{V}_n$ . . . . *Sequencer*

Interface Operations `console`: $\mathcal{I} \longrightarrow [\,\mathcal{C}_0\ \mathcal{C}_1\ \cdots\,]$ . . . . *Interactive input*

`screen`: $[\,\mathcal{C}_0\ \mathcal{C}_1\ \cdots\,] \longrightarrow [\,]$ . . . . *Interactive output*
`dski`: $\mathcal{I} \longrightarrow [\,\mathcal{C}_0\ \mathcal{C}_1\ \cdots\,]$ . . . . *File input*
`dsko`: $[\,\mathcal{I}[\,\mathcal{C}_0\ \mathcal{C}_1\ \cdots\,]\,] \longrightarrow [\,]$ . . . . *File output*

Text Generation `issue`: $\mathcal{V} \longrightarrow [\,\mathcal{C}_0\ \mathcal{C}_1\ \cdots\,]$ . . . . *Generate text*

Scanning `scan`: $[\,\mathcal{C}_0\ \mathcal{C}_1\ \cdots\,] \longrightarrow [\,\mathcal{A}\ \mathcal{C}_j\ \mathcal{C}_{j+1}\ \cdots\,]$ . . . . *Scan text*

`scans`: $[\,\mathcal{C}_0\ \mathcal{C}_1\ \cdots\,] \longrightarrow [\,\mathcal{A}_0\ \mathcal{A}_1\ \cdots\,]$ . . . . . `scan` *iterated*

Parsing `parse`: $[\,\mathcal{C}_0\ \mathcal{C}_1\ \cdots\,] \longrightarrow [\,\mathcal{E}_0\ \mathcal{E}_1\ \cdots\,]$ . . . . . `xparses` $\circ$ `scans`

`xparse`: $[\,\mathcal{T}_0\ \mathcal{T}_1\ \cdots\,] \longrightarrow [\,\mathcal{E}\ \mathcal{T}_j\ \mathcal{T}_{j+1}\ \cdots\,]$ . . . . . *Parse text*
`xparses`: $[\,\mathcal{T}_0\ \mathcal{T}_1\ \cdots\,] \longrightarrow [\,\mathcal{E}_0\ \mathcal{E}_1\ \cdots\,]$ . . . . . `parse` *iterated*

Special Operations `val`: $\mathcal{E} \longrightarrow \mathcal{V}$ . . . . *Evaluate*

`evlst`: $[\,\mathcal{E}_0\ \mathcal{E}_1\ \cdots\,] \longrightarrow [\,\mathcal{V}_0\ \mathcal{V}_1\ \cdots\,]$ . . . . . . `val` *iterated.*
`let`: $[\,\mathcal{X}\ \mathcal{E}_1\ \mathcal{E}_2\,] \longrightarrow \mathcal{V}$ . . . . *Lexical binder*
`rec`: $[\,\mathcal{X}\ \mathcal{E}_1\ \mathcal{E}_2\,] \longrightarrow \mathcal{V}$ . . . . *Recursive binder*
`fix`: $[\,\mathcal{X}\ \mathcal{E}\,] \longrightarrow \mathcal{V}$ . . . . *Recursive binder*

[

]**REFERENCE (b) – Daisy Operations**

Daisy Operations
(by group)

Types in descriptors		
	$\mathcal{B}$ – `T` *or* `[]`	
$\mathcal{N}$ – **a numeral**	$\mathcal{L}$ – **a list**	$\mathcal{C}$ – **a character**
$\mathcal{I}$ – **a literal**	$\mathcal{A}$ – **an atom**	$\mathcal{E}$ – **an expression**
	$\mathcal{V}$ – **a value**	

`add`: `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow \mathcal{N}_3$ . . . . *Add*

`all?`: `[` $\mathcal{V}_0$ $\mathcal{V}_1$ $\cdots$ `]` $\longrightarrow \mathcal{B}$ . . . . *Test for no null elements*

`and`: `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow \mathcal{N}_3$ . . . . *Logical-and*

`any?`: `[` $\mathcal{V}_0$ $\mathcal{V}_1$ $\cdots$ `]` $\longrightarrow$ `[` $\mathcal{V}_i$ $\mathcal{V}_{i+1}$ $\cdots$ `]` . *Locate a non-Nil element*

`asAPL`: $\mathcal{V} \longrightarrow \mathcal{A}$ . . . . *Cite as an application*

`asDCT`: $\mathcal{V} \longrightarrow \mathcal{D}$ . . . . *Cite as a directive*

`asERR`: $\mathcal{V} \longrightarrow \mathcal{M}$ . . . . *Error*

`asFTN`: $\mathcal{V} \longrightarrow \mathcal{F}$ . . . . *Cite as a function*

`asIDE`: $\mathcal{V} \longrightarrow \mathcal{I}$ . . . . *Cite as an identifier*

`asLST`: $\mathcal{V} \longrightarrow \mathcal{L}$ . . . . *Cite as a list*

`asNML`: $\mathcal{V} \longrightarrow \mathcal{N}$ . . . . *Cite as a numeral*

`Chr?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Character test*

`ChrAsNml`: $\mathcal{C} \longrightarrow \mathcal{N}$ . . . . *Character's numeric code*

`console`: $\mathcal{I} \longrightarrow$ `[` $\mathcal{C}_0$ $\mathcal{C}_1$ $\cdots$ `]` . . . . *Interactive input*

`cons`: `[` $\mathcal{V}_1$ $\mathcal{V}_2$ `]` $\longrightarrow$ `[` $\mathcal{V}_1$ `!` $\mathcal{V}_2$ `]` . . . . *List constructor*

`crc_hd`: $\mathcal{V} \longrightarrow \mathcal{V}$ . . . . *Coerce-head*

`crc_tl`: $\mathcal{V} \longrightarrow \mathcal{V}$ . . . . *Coerce-tail*

`dcr`: $\mathcal{N}_1 \longrightarrow \mathcal{N}_2$ . . . . *Decrement*

`div`: `[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\longrightarrow \mathcal{N}_3$ . . . . *Divide*

`dski`: $\mathcal{I} \longrightarrow$ `[` $\mathcal{C}_0$ $\mathcal{C}_1$ $\cdots$ `]` . . . . *File input*

`dsko`: $[\,\mathcal{I}[\,\mathcal{C}_0\ \ \mathcal{C}_1\ \cdots\,]\,] \longrightarrow$ `[]` . . . . *File output*

`eq?`: $[\,\mathcal{N}_1\ \ \mathcal{N}_2\,] \longrightarrow\ \mathcal{B}$ . . . . *Numeric equality*

`evlst`: $[\,\mathcal{E}_0\ \ \mathcal{E}_1\ \cdots\,] \longrightarrow [\,\mathcal{V}_0\ \ \mathcal{V}_1\ \cdots\,]$ . . . . . . `val` *iterated.*

`fix`: $[\,\mathcal{X}\ \ \mathcal{E}\,] \longrightarrow \mathcal{V}$ . . . . *Recursive binder*

`frons`: $[\,\mathcal{V}\ \ \mathcal{V}'\,] \longrightarrow \mathcal{L}$ . . . . *Multiset constructor*

`ge?`: $[\,\mathcal{N}_1\ \ \mathcal{N}_2\,] \longrightarrow\ \mathcal{B}$ . . . . *At-least*

`gt?`: $[\,\mathcal{N}_1\ \ \mathcal{N}_2\,] \longrightarrow\ \mathcal{B}$ . . . . *Greater-than*

`head`: $[\,\mathcal{V}_1$ `!` $\mathcal{V}_2\,] \longrightarrow \mathcal{V}_1$ . . . . *Head of a list*

`_hd`: $\mathcal{V} \longrightarrow \mathcal{V}'$ . . . . *Head of a composite object*

`if`: $[\,\mathcal{T}_0\ \ \mathcal{V}_0\ \ \mathcal{T}_1\ \ \mathcal{V}_1\ \cdots\,] \longrightarrow \mathcal{V}_i$ . . . . . *Conditional operation*

`in?`: $[\,\mathcal{U}\ \ [\,\mathcal{V}_0\ \ \mathcal{V}_1\ \cdots\,]\,] \longrightarrow \mathcal{B}$ . . . . *Membership Test*

`inc`: $\mathcal{N}_1 \longrightarrow\ \mathcal{N}_2$ . . . . *Increment*

`inv`: $\mathcal{N}_1 \longrightarrow\ \mathcal{N}_2$ . . . . *Invert*

`isAPL?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a application-object*

`isAtm?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for an Atom*

`isDCT?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a directive*

`isERR?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for an identifier-object*

`isFTN?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a function-object*

`isIDE?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a identifier-object*

`isLST?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a list-object*

`isLST?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a list*

`isLtrl?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a literal*

`isNML?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a numeral*

`isNML?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for a numeral*

`issue`: $\mathcal{V} \longrightarrow [\,\mathcal{C}_0\ \ \mathcal{C}_1\ \cdots\,]$ . . . . *Generate text*

`le?`: $[\,\mathcal{N}_1\ \ \mathcal{N}_2\,] \longrightarrow\ \mathcal{B}$ . . . . *At-most*

`let`: $[\,\mathcal{X}\ \ \mathcal{E}_1\ \ \mathcal{E}_2\,] \longrightarrow \mathcal{V}$ . . . . *Lexical binder*

`lt?`: $[\,\mathcal{N}_1\ \ \mathcal{N}_2\,] \longrightarrow\ \mathcal{B}$ . . . . *Less-than*

`mpy`: $[\,\mathcal{N}_1\ \ \mathcal{N}_2\,] \longrightarrow \mathcal{N}_3$ . . . . *Multiply*

`ne?`: $[\,\mathcal{N}_1\ \ \mathcal{N}_2\,] \longrightarrow\ \mathcal{B}$ . . . . *Numeric inequality*

`neg?`: $\mathcal{N} \longrightarrow \mathcal{B}$ . . . . *Test for negative*

`neg`: $\mathcal{N}_1 \longrightarrow \mathcal{N}_2$ . . . . *Negate*

`nil?`: $\mathcal{V} \longrightarrow \mathcal{B}$ . . . . *Test for Nil*

`NmlAsChr`: $\mathcal{N} \longrightarrow \mathcal{C}$ . . . . *Convert a numeral to a character*

one?: $\mathcal{N} \longrightarrow \mathcal{B}$ . . . . *Test for One*

or: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Logical-or*

parse: [$\mathcal{C}_0$ $\mathcal{C}_1$ $\cdots$] $\longrightarrow$ [$\mathcal{E}_0$ $\mathcal{E}_1$ $\cdots$] . . . . . xparses $\circ$ scans

pos?: $\mathcal{N} \longrightarrow \mathcal{B}$ . . . . *Test for positive*

rec: [$\mathcal{X}$ $\mathcal{E}_1$ $\mathcal{E}_2$] $\longrightarrow \mathcal{V}$ . . . . *Recursive binder*

rem: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Remainder*

same?: [$\mathcal{U}$ $\mathcal{V}_1$ $\cdots$ $\mathcal{V}_n$] $\longrightarrow \mathcal{B}$ . . . . *Reference equality*

scans: [$\mathcal{C}_0$ $\mathcal{C}_1$ $\cdots$] $\longrightarrow$ [$\mathcal{A}_0$ $\mathcal{A}_1$ $\cdots$] . . . . . . scan *iterated*

scan: [$\mathcal{C}_0$ $\mathcal{C}_1$ $\cdots$] $\longrightarrow$ [$\mathcal{A}$ $\mathcal{C}_j$ $\mathcal{C}_{j+1}$ $\cdots$] . . . . *Scan text*

ScnCTL?: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Control-character test*

ScnDGT?: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Digit-character test*

ScnLFA?: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Alpha-character test*

ScnNON?: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Neutral-character test*

ScnSPC?: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Space-character test*

ScnSYM?: $\mathcal{C} \longrightarrow \mathcal{B}$ . . . . *Symbol-character test*

screen: [$\mathcal{C}_0$ $\mathcal{C}_1$ $\cdots$] $\longrightarrow$ [] . . . . *Interactive output*

seq: [$\mathcal{V}_0$ $\mathcal{V}_1$ $\cdots$ $\mathcal{V}_n$] $\longrightarrow \mathcal{V}_n$ . . . . *Sequencer*

sgn: $\mathcal{N} \longrightarrow$ -1 or 1 . . . . *Sign projection*

sub: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Subtract*

TagOf: $\mathcal{V} \longrightarrow \mathcal{N}$ . . . . *Numeric value of a tag*

tail: [$\mathcal{V}_1$! $\mathcal{V}_2$] $\longrightarrow \mathcal{V}_1$ . . . . *Tail of a list*

_tl: $\mathcal{V} \longrightarrow \mathcal{V}'$ . . . . *Tail of a composite object*

val: $\mathcal{E} \longrightarrow \mathcal{V}$ . . . . *Evaluate*

xor: [$\mathcal{N}_1$ $\mathcal{N}_2$] $\longrightarrow \mathcal{N}_3$ . . . . *Exclusive-or*

xparses: [$\mathcal{T}_0$ $\mathcal{T}_1$ $\cdots$] $\longrightarrow$ [$\mathcal{E}_0$ $\mathcal{E}_1$ $\cdots$] . . . . . . parse *iterated*

xparse: [$\mathcal{T}_0$ $\mathcal{T}_1$ $\cdots$] $\longrightarrow$ [$\mathcal{E}$ $\mathcal{T}_j$ $\mathcal{T}_{j+1}$ $\cdots$] . . . . . . *Parse text*

zero?: $\mathcal{N} \longrightarrow \mathcal{B}$ . . . . *Test for Zero*

[

]**REFERENCE (c) – Daisy Operations**

Arithmetic Operations

`add:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{N}_3$	**add**	$\mathcal{N}_3 = \mathcal{N}_1 + \mathcal{N}_2$
`sub:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{N}_3$	**subtract**	$\mathcal{N}_3 = \mathcal{N}_1 - \mathcal{N}_2$
`div:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{N}_3$	**divide**	$\mathcal{N}_3 = \mathcal{N}_1 \div \mathcal{N}_2$
`mpy:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{N}_3$	**multiply**	$\mathcal{N}_3 = \mathcal{N}_1 \cdot \mathcal{N}_2$
`rem:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{N}_3$	**remainder**	$\mathcal{N}_3 = \mathcal{N}_1 - (\mathcal{N}_1 \div \mathcal{N}_2) \cdot \mathcal{N}_2$
`inc:` $\mathcal{N}$ $\rightarrow$ $\mathcal{N}'$	**increment**	$\mathcal{N}' = \mathcal{N} + 1$
`dcr:` $\mathcal{N}$ $\rightarrow$ $\mathcal{N}'$	**decrement**	$\mathcal{N}' = \mathcal{N} - 1$
`neg:` $\mathcal{N}$ $\rightarrow$ $\mathcal{N}'$	**negate**	$\mathcal{N}' = -\mathcal{N}$
`inv:` $\mathcal{N}$ $\rightarrow$ $\mathcal{N}'$	**invert**	$\mathcal{N}' = \overline{\mathcal{N}}$
`sgn:` $\mathcal{N}$ $\rightarrow$ $\mathcal{N}'$	**sign**	$\mathcal{N}' = \begin{cases} \texttt{1}, & \textbf{if } \mathcal{N} \geq 0 \\ \texttt{-1}, & \textbf{if } \mathcal{N} < 0 \end{cases}$
`and:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{N}_3$	**binary-and**	$\mathcal{N}_3 = \mathcal{N}_1 \odot \mathcal{N}_2$
`or:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{N}_3$	**binary-or**	$\mathcal{N}_3 = \mathcal{N}_1 \oplus \mathcal{N}_2$
`xor:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{N}_3$	**exclusive-or**	$\mathcal{N}_3 = \mathcal{N}_1 \otimes \mathcal{N}_2$
`zero?:` $\mathcal{N}$ $\rightarrow$ $\mathcal{B}$	**is-zero?**	$\mathcal{N} = \texttt{0}$ **?**
`one?:` $\mathcal{N}$ $\rightarrow$ $\mathcal{B}$	**is-one?**	$\mathcal{N} = \texttt{1}$ **?**
`neg?:` $\mathcal{N}$ $\rightarrow$ $\mathcal{B}$	**negative?**	$\mathcal{N} < \texttt{0}$ **?**
`pos?:` $\mathcal{N}$ $\rightarrow$ $\mathcal{B}$	**positive?**	$\mathcal{N} \geq \texttt{0}$ **?**
`lt?:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{B}$	**less-than**	$\mathcal{N}_1 < \mathcal{N}_2$ **?**
`le?:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{B}$	**at-most**	$\mathcal{N}_1 \leq \mathcal{N}_2$ **?**
`eq?:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{B}$	**equal**	$\mathcal{N}_1 = \mathcal{N}_2$ **?**
`ne?:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{B}$	**unequal**	$\mathcal{N}_1 \neq \mathcal{N}_2$ **?**
`ge?:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{B}$	**at-most**	$\mathcal{N}_1 \geq \mathcal{N}_2$ **?**
`gt?:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{B}$	**greater-than**	$\mathcal{N}_1 > \mathcal{N}_2$ **?**
`lt?:[` $\mathcal{N}_1$ $\mathcal{N}_2$ `]` $\rightarrow$ $\mathcal{B}$	**less-than**	$\mathcal{N}_1 < \mathcal{N}_2$ **?**

Chapter 4

Errons

There are no provisions in Daisy to handle erroneous or exceptional conditions. A detected error produces a distinguished value, called and *erron,* which records the occurance. Errons are simply a class of values produced by the Daisy interpreter. They appear as messages, surrounded by '`|`' characters. A typical example is

`|ubi:RED|` ,

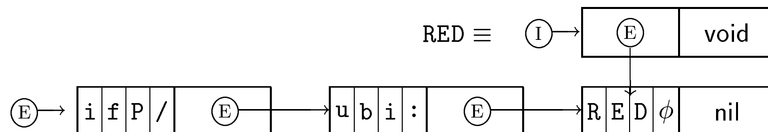
which says, "`RED` is an unbound identifier."

Errons cannot be manipulated by Daisy programs. In interpretation, they are treated like discovered divergences, or detected instances of the "undefined" value. That is, the presence of an erron is treated as though the value were $\perp$, where that makes sense. Any program attempting to manipulate an erroneous/undefined a value, becomes erroneous/undefined itself.

The displayed text is diagnostic, and errons accumulate text as their use propagates. For example, if `N` is unbound, then the expression `if:[RED "stop" "go"]` is also erroneous, and would produce the erron

`|ifP/ubi:RED|`

The '`/`' can be read as, "due to." The message above says, "`If` has an invalid predication, due to the unbound identifier, `RED`." An erron is represented as a sequence of unary cells,



A literal's name is simply an erron in a distingished position, as illustrated above. The expression `(\[X].X):"RED"` evaluates to `RED`'s head or the erron `|RED|`.

The messages accumulates three-letter prefixes, noting compound errors. The first section below explains these prefixes. The final suffix can be nothing, the name of a literal, or a *syntax-erron* (discussed below), depending on the circumstances. The errouneous expression `inc:[5]` evaluates to

`|nn0/|` ,

while `inc:"RED"` yields

`|nn0/RED|` .

In both cases, the prefix '`nn0`' reports a nonnumeric argument—`inc` expects a numeral. The text '`RED`' is incorporated because a literal's name enjoys a compatable representation, while a structure, such as `[5]` is not readily rendered into message text.

Daisy's parsing operations develop a language of *syntax errons,* described in the second section below. In interactive programming, these values are conveyed in the expression stream to the interpreter.

4.1 Prefixes

|all| *Erroneous list (*all?*).*

The argument to the all? operation either is not a list, or contains an erron.

|any| *Erroneous list (*any?*).*

The argument to the any? operation either is not a list, or contains an erron.

|arg| *Invalid formal argument.*

The formal argument of a function contains something other than literals and lists. A common cause is the mistaken use of angle delimiters in a function expression. For example, the expression (\[X].X):[5] produces |arg/X|. Perhaps (\[X].X):[5] was intended; [X] is represented as an application and cannot be used as a formal argument. However, its occurance is not discovered by the parser; it is detected during interpretation. The message suffix often records the name of identifer whose binding is sought.

NOTE**[March 11, 1996]: This error can also arise if locating a binding requires more than 255 successive tail-operations. This is due to a known problem in implementation, which is noted for revision.**

|chr| *Non-character operand.*

The operand to one of the character manipulation operations is not a character. The operation involved is one of Chr?, ChrAsNml, ScnCTL?, ScnSPC?, ScnDGT?, ScnLFA?, ScnNON?, or ScnSYM?. In Daisy, characters are simply literals whose name consists of a single character code. The Chr? operation tests for this condition.

|cmp| *Erroneous comparison (*same? *or* in?*).*

The same? and in? operations use the same primitive test for reference equality. The comparison has failed because one of the operands is an erron.

|crc| *Invalid coercion.*

The operand to one of asDCT, asNML, asFTN, asAPL, asLST, or asIDE either is either erroneous or invalid. An operand is invalid when changing the tag of its citation would violate a storage classification. The table below shows what coercions are permitted (See also the reference operations in the Section Operations).

	← Coercion →						
	DCT	NML	IDE	LST	APL	FTN	ERR
DCT	*ok*	*ok*	*err*	*err*	*err*	*err*	*err*
NML	*ok*	*ok*	*err*	*err*	*err*	*err*	*err*
IDE	*err*	*err*	*ok*	*ok*	*ok*	*ok*	*err*
LST	*err*	*err*	*ok*	*ok*	*ok*	*ok*	*err*
APL	*err*	*err*	*ok*	*ok*	*ok*	*ok*	*err*
FTN	*err*	*err*	*ok*	*ok*	*ok*	*ok*	*err*
ERR	*err*	*err*	*err*	*err*	*err*	*err*	*err*

|dfn| *Invalid assignment.*

The operand to the assignment operation must have the form

[*Identifier* ! *Value*]

This erron results if thing assigned is not an identifier. It is not the result of a syntax error, which would develop a parsing erron of the form |...@'='|. The 'dfn/' prefix is due to a poorly *constructed* **assignment command. Use of the assignment mechanism is** *not recommended.* **Strongly. There is no assurance that any manner of assignment will be supported in future versions of the language.**

|dvc| *Device error.*

This prefix indicates a problem at the host input-output interface. The operation involved is one of console, dski, dsko, or screen. The conditions that raise the 'dvc/' prefix are

an erroneous operand or failure to open the host file. For example, if `F` is unbound, the expression `dski:F` has value

`|dvc/ubi:F|` ;

and if no file named `F` exists, `dski:"F"` has value

`|dvc|` .

In the latter case, the DSI system also issues a diagnostic query:

```
query:  [H] Can't put in
```

A similar `query`-message is displayed when there is any problem with actual input or output.

`|evl|` *Non-list argument (*`evlst`*).*

The `evlst` operation is specified as

```
evlst = ^\L.
  if:[ nil?:L
         []
       isLST?:L
         [ val:head:L ! evlst:tail:L]
       "otherwise"
         ERROR
      ]
```

This prefix reports that `evlst` has reached the point *ERROR* above.

`|f-c|` *Construction error*

When a list is applied, the construction functional expects the argument to have the form

$$[\,L_0\;\; L_1\;\; \cdots\;\; L_n\,] \quad ,$$

where each L_i is a list. The '`f-c`' prefix indicates that a an invalid structure is detected prior to invoking transposition (See the '`xps`' prefix). Either the argument itself is not a list, or in the case that a cyclic list is applied, L_0 is not a list.

`|ftn|` *Erron applied.*

An error has occurred in the function part of an application. If `F` is unbound, the expression `F:[5]` evaluates to

`|ftn/ubi:F|`

`|hd?|` *Invalid head-access*

This prefix is developed by the primitive list processing system. It indicates an attempt to access the head of an object that does not have a "head" field. The operand may be a directive. If not, the object cited may be an erron, a printname, or a numeral. The '`hd?`' prefix arises when the `_hd` operation is used instead of `head`. The more primitive `_hd` is used for environment look-up; hence this prefix usually reflects a mismatch between formal arguments and actual arguments. For example, the expression `(\.[X].X):5` yields a `|hd?|` erron because `X` is bound to `5`'s nonexistent head.

NOTE **[March 11, 1996]: The addition of more verification in environment look-up is under consideration**

`|ifA|` *Invalid alternative (`if`).*

The operand to the `if` operation does not have the expected structure,

$$[\, p_0 \;\; v_0 \;\; p_1 \;\; v_1 \;\; \cdots \; p_n \;\; v_n \;\; v_{n+1} \,]$$

This prefix says nothing about the selected alternative, v_i. For example, if `N` is unbound, the expression `if:[[] 5 N]` has value `|ubi:N|`; `if` does not verify the value it chooses. The '`ifa`' arises `if` encounters something other than a list, as in the expression `if:"A"`.

NOTE **[March 11, 1996]: An expression like `if:[[] X]` returns Nil's head, #.**

`|ifP|` *Erroneous predication (`if`).*

One of the tests, p_i, in `if`'s argument,

$$[\, p_0 \;\; v_0 \;\; p_1 \;\; v_1 \;\; \cdots \; p_n \;\; v_n \;\; v_{n+1} \,] \quad ,$$

is an erron.

`|ld?|` *Invalid access*

This prefix is equivalent to "`hd?` or `tl?`."

`|lst|` *Erron in list-expression.*

The implicit operation associated with delimiters '`[`' and '`]`' has encountered an erron. This prefix is unlikely to occur unless a program is used to build the offending expression. Below, `LSTdct` is assigned the directive associated with the list primitive. `LSTexp` builds a list expression; for instance, `LSTexp:^[a b c]` builds the *expression* `[a b c]`.

```
LSTdct = head:asLST:^[anything]
LSTexp = ^\L. asAPL:[LSTdct ! L]
```

Now, if `X` is unbound, `LSTexp:[X "Y"]` yields `[|ubi:X| Y]`. It is impossible to obtain such a form *via* Daisy's parsing operations. Should this expression be evaluated, say by `val:LSTexp:[X "Y"]`, the result is

```
[ |lst/ubi:X| |ubi:Y| ]
```

The '`lst`' prefix records the error in expression formation.

`|opn|` *Invalid operation code.*

A directive is applied, for which no primitive operation is associated. This condition does not arise in casual programming; it is the result either of a mistaken program-building program, or the mistaken use of directives as data values (This is *not* recommended). The expression `(asDCT:66):5` yields `6` because the `.66` directive encodes the `inc` operation [as of March 11, 1996]. The expression `(asDCT:200):5` yields the erron `|opn|` because there is no operation whose encoding is `.200`.

`|nla|` *Non-list operand (* `head` *or* `tail` *).*

The operand to the `head` or the `tail` operation is not a list.

`|nn0|` *Non-numeric operand.*

The operand to a unary arithmetic operation is not a numeral, or the second operand to a binary arithmetic operation is not a numeral.

In the first case, the error occured in the applying `dcr`, `inc`, `inv`, `neg`, `neg?`, `NmlAsChr`, `one?`, `pos?`, `sgn`, **or** `zero?`.

The second case involves one of `add`, `and`, `div`, `eq?`, `ge?`, `gt?`, `lt?`, `le?`, `mpy`, `ne?`, `or`, `rem`, `sub`, **or** `xor`.

The error is also raised when the operand to the `%settrc` **operation (used in implementation development) is invalid.**

`|nn1|` *Non-numeric operand.*

The second operand to a binary arithmetic operation is not a numeral. The error involves one of the operations `add`, `and`, `div`, `eq?`, `ge?`, `gt?`, `lt?`, `le?`, `mpy`, `ne?`, `or`, `rem`, `sub`, **or** `xor`.

`|prb|` *Invalid numeric probe.*

The offending expression involves the application of a numeral. Either the applied numeral is outside the range from `0` **to** `16777215`**, or the argument is not a list, or the probe exceeds the length of the argument. The upper limit of** `16777215` **($2^{24} - 1$) on numeric probes is imposed for compatability with certain data representations.**

`|prs|` *Invalid argument (*`parse`*,* `xparse`*,* `xparses`*).*

The `parse`, `xparse` **and** `xparses` **operations expect an argument of the form**

$$[\, v_0 \;\; v_1 \;\; \cdots \,] \quad .$$

This prefix results when a non-list tail is encountered, or when some v_i is an erron.

`|sam|` *Invalid argument (*`same?` *or* `in?`*).*

The `same?` operation expects an argument of the form

$$[\, u \;\; v_0 \;\; v_1 \;\; \cdots \;\; v_n \,] \quad .$$

It returns `T` if u is identical to any of the values v_i. The expression `in?:`<u L> is like `same?:`<u `!` L>. The 'sam' prefix arises when the argument is discovered to have a different structure.

`|scn|` *Invalid argument (*`scan`*,* `scans`*).*

The argument to the `scan` or `scans` operation is not a list.

`|sc0|` *Invalid text (*`scan`*,* `scans`*).*

The scanning operations expect an argument of the form

$$[\, v_0 \;\; v_1 \;\; \cdots \;\; v_n \,] \quad .$$

This prefix results when a non-list tail is encountered, or when some v_i is an erron.

`|sc1|` *Erroneous text (*`scan`*,* `scans`*).*

This prefix indicates that an erron is discovered in the scanned text.

`|seq|` *Invalid argument (*`seq`*).*

The `seq` operation expects an argument of the form

$$[\, v_0 \;\; v_1 \;\; \cdots \;\; v_n \,] \quad .$$

The '`seq`' prefix arises when the argument is discovered to have a different structure.

`|tag|` *Invalid citation.*

The operand to one of `nil?`, `isLtrl?`, `isAtm?`, `isDCT?`, `isNML?`, `isFTN?`, `isAPL?`, `isLST?`, or `isIDE?` is erroneous.

`|tl?|` *Invalid tail-access*

This prefix is developed by the primitive list processing system. It indicates an attempt to access the tail of an object that does not have a "tail" field. The operand is a directive because all other objects used by Daisy have tails. The '`tl?`' prefix arises when the `_tl` operation is used instead of `tail`. The more primitive `_tl` is used for environment look-up; hence this prefix often reflects a mismatch between formal arguments and actual arguments. Since almost every object has a tail, '`tl?`' isn't often seen. One example is the expression `(\.[X Y ! Z].Z):"A"` returing the tail of `A`'s tail. Assuming `A` is unassigned, its tail is the `void` directive; thus the result is the erron `|tl?|`.

NOTE **[March 11, 1996]: The addition of more verification in environment look-up is under consideration**

`|trj|` *Invalid trajectory.*

A *trajectory* represents the path locating the binding of an indentifier in the environment. Since this is an internally generated representation [as of March 11, 1996], this prefix indicates an *implementation error* and not a programming error.

`|ubi:|` *Unbound identifier.*

An identifier has no binding—or associated value—in the evaluation environment. Where the identifier is a literal its name is incorporated as a suffix. For example, in an empty environment, the expression `[1 green toad]` evaluates to

```
[1  |ubi:green|  |ubi:toad|]
```

`|val|` *Value-of-erron.*

The argument to the `val` operation is erroneous. This prefix typically arises when a syntax error has occurred on input. Parsing produces an erron reporting the bad syntax, which is passed through to interpretation. For instance, if the operator types '`[1 2 3]`', the response is

```
|val/[nnn@‘]’|   ,
```

indicating an error in evaluation, due to the occurance of an unbalanced ']' (See the section below on syntax errons). Mistaken use of angle delimiters in binding forms is another cause for this prefix. If `X` is unbound, the expression `let:[X 5 X]` yields

```
|val/ubi:X|
```

because `let`'s actual argument is

```
[ |ubi:X|  5  |ubi:X| ]
```

Often, the form `let:[X 5 X]` was intended.

`|xps|` *Transposition error.*

When a list is applied, the construction functional expects the argument to have the form

$$[\, L_0 \;\; L_1 \;\; \cdots \;\; L_n \,] \quad ,$$

where each L_i is a list. This structure is implicitly transposed, and the 'xps' prefix indicates a problem detected during this phase of construction. It means that a non-list is encountered at the surface structure in transposition; for instance, a form like

```
[(\x.x)*]:[[1 2] !  BAD]
```

yields

```
[ [ 1 ! |xps/ubi:BAD| ]  [ 2 ! |xps/xps/ubi:BAD| ] ]
```

when `BAD` is unbound. As above, it is usually the case that the argument holds an erroneous value. See also the prefix '`f-c`'.

4.2 Syntax Errons

The `parse` and `parses` operations develop a language of errons reporting syntax errors. They record the parsing state to the point that the error is detected. The suffix of a parsing erron typically looks like

$$|\text{—}@\text{‘}c\text{’}| \quad ,$$

where c is the character at which parsing fails. For example, the expression `xparse:[ "]" ]` yields the erron `|@‘]’|` because ‘]’ is an unbalanced right-hand delimiter. However, there are cases where the point of failure is not a character. When parsing fails at an erron, the erron's text is appended. For example, the expression `xparse:[ BAD ]` produces the erron

```
|..@/ubi:BAD|
```

When parsing fails at a form, a letter designates the failure. For example, the expression `xparse:["\" "X" 5]` produces the erron

```
|.\@n|
```

because a numeral, indicated by ‘n’, is seen where a ‘.’ is expected after the X. Here and below, a lower-case letter reports a non-character object, either some unexpected input or a partial result.

Indicator	Object
d	a directive
n	a numeral
m	an erron
i	an identifer object
l	a list object
a	an application object
f	a function object

The leading text records the state of parsing up to the point of

failure. Let o be ‘d’, ‘n’, ‘m’, ‘i’, ‘l’, ‘a’, or ‘f’; and let c be the point of failure.

Message	**Explanation**
.—@‘*c*’	**Message fill**
..—@‘*c*’	**Message fill**
...—@‘*c*’	**Message fill**
^—@‘*c*’	**Error in value quotation**
(—@‘*c*’	**Error in parenthesization**
o:—@‘*c*’	**Error in application’s argument**
\—@‘*c*’	**Error in formal argument**
o.—@‘*c*’	**Error in function body**
[—@‘*c*’	**Error before matching ‘ ···]’**
[—@‘*c*’	**Error before matching ‘ ···]’**
{—@‘*c*’	**Error before matching ‘ ··· }’**
!—@‘*c*’	**Error after a ‘!’**
@‘=’	**Error (nonliteral) before ‘=’**
o=—@‘*c*’	**Error after an ‘=’**

Some parsing errons are illustrated below.

<table>
<tr><th>Input text</th><th>Erron</th><th>Note</th></tr>
<tr><td>.</td><td>|@‘.’|</td><td>1</td></tr>
<tr><td>[.</td><td>|...[@‘.’|</td><td>2</td></tr>
<tr><td>^.</td><td>|...^@‘.’|</td><td>3</td></tr>
<tr><td>^[.</td><td>|..^[@‘.’|</td><td>4</td></tr>
<tr><td>[a !.</td><td>|.[i!@‘.’|</td><td>5</td></tr>
<tr><td>(.</td><td>|...(@‘.’|</td><td>6</td></tr>
<tr><td>\[x.</td><td>|.\[i@‘.’|</td><td>7</td></tr>
<tr><td>\x.[a.</td><td>|...\i.[i@‘.’|</td><td>8</td></tr>
<tr><td>^\x.[a.</td><td>|..^\i.[i@‘.’|</td><td>9</td></tr>
<tr><td>f:[a.</td><td>|i:[i@‘.’|</td><td>10</td></tr>
</table>

NOTES

1. **It is an error for a ‘.’ to occur anywhere other than between the formal argument and body of a function expression. This fact is used in the remaining examples.**

2. The error is an unbalanced '`[`'; the leading periods are fill characters in parsing errons.

3. This example shows an error in quotation.

4. This example shows an error in forming a quoted list. In general, the parser develops a message showing what forms are in progress at the point of an error.

5. Here, an error occurs in forming the tail of a list expression.

6. Here, an error within parentheses.

7. Here, the error is in the formal argument of a function expression.

8. Here, the error is in the body of a function expression. The '`i`' in the message indicates that the function's formal argument is an identifier.

9. This is the same error as in Example 8, but the function is also quoted.

10. The error is in the argument part of an application expression. The '`i`' indicates that the function part is an identifier.

    The purpose below is to show what class of object is created for kinds of Daisy expressions. The description of the `parse` operation and the section on Daisy interpretation give details of expression representation.

Input text	Erron	Note
`[ 2 .`	`|..[n@'.'|`	11(a)
`[ A .`	`|..[i!@'.'|`	11(b)
`[ ^A .`	`|..[i!@'.'|`	11(c)
`[ (A) .`	`|..[a!@'.'|`	11(d)
`[ F:X .`	`|..[a!@'.'|`	11(e)
`[ \X.X .`	`|..[f!@'.'|`	11(f)
`[ [A] .`	`|..[l!@'.'|`	11(g)
`[ [A] .`	`|..[a!@'.'|`	11(h)
`[ {A}.`	`|..[a!@'.'|`	11(i)
`[ [] .`	`|..[i!@'.'|`	11(j)

NOTES

11. **In each case the error is in the formation of a list expression, but the error occurs after an element-expression has been successfully parsed.**

    (a) **The numeral `2` is indicated by '`n`'.**

    (b) **The literal `A` is indicated by '`i`'; literals are represented in identifer cells.**

    (c) **Both value and literal quotations are also forms of identification; hence, the quotation `^A` is indicated by an '`i`'.**

    (d) **A parenthesized expression is a form of application and is indicated by an '`a`'.**

    (e) **The application expression `f:a` is indicated by an '`a`'.**

    (f) **The function expression `\X.X` is indicated by an '`f`'.**

    (g) **The list expression `[x]` forms a true list object, indicated by an '`l`'.**

    (h) **The form `[x]` is headed by an application cell, applying the implicit list opertation. Thus, this form is indicated by an '`a`'.**

    (i) **As in case (h), the form `{x}` abbreviates application of the implicit set operation. It too is indicated by an '`a`'.**

    (j) **Nil is an identifier—hence the '`i`'—and the forms `[]` and `{}` are resolved to Nil during parsing.**

Errons

`|—any/—|` *Erroneous list (* `all?` *).*

`|—any/—|` *Erroneous list (* `any?` *).*

`|—arg/—|` *Invalid formal argument.*

`|—chr/—|` *Non-character operand.*

`|—cmp/—|` *Erroneous comparison (* `same?` *or* `in?` *).*

`|—crc/—|` *Invalid coercion.*

`|—dfn/—|` *Invalid assignment.*

`|—dvc/—|` *Device error.*

`|—evl/—|` *Non-list argument (* `evlst` *).*

`|—f-c/—|` *Construction error*

`|—ftn/—|` *Erron applied.*

`|—hd?/—|` *Invalid head-access*

`|—ifA/—|` *Invalid alternative (* `if` *).*

`|—ifP/—|` *Erroneous predication (* `if` *).*

`|—ld?/—|` *Invalid access*

`|—lst/—|` *Erron in list-expression.*

`|—opn/—|` *Invalid operation code.*

`|—nla/—|` *Non-list operand (* `head` *or* `tail` *).*

`|—nn0/—|` *Non-numeric operand.*

`|—nn1/—|` *Non-numeric operand.*

`|—prb/—|` *Invalid numeric probe.*

`|—sam/—|` *Invalid argument (* `same?` *or* `in?` *).*

`|—scn/—|` *Invalid argument (* `scan`*,* `scans` *).*

`|—seq/—|` *Invalid argument (* `seq` *).*

`|—scn/—|` *Invalid argument (* `parse`*,* `xparse`*,* `xparses` *).*

`|—sc0/—|` *Invalid text (* `scan`*,* `scans` *).*

`|—sc1/—|` *Erroneous text (* `scan`*,* `scans` *).*

`|—tag/—|` *Invalid citation.*

`|—tl?/—|` *Invalid tail-access*

`|—trj/—|` *Invalid trajectory.*

`|—ubi:—|` *Unbound identifier.*

`|—val/—|` *Value-of-erron.*

`|—xps/—|` *Transposition error.*

Syntax Errons

Syntax Erron Indicators

Indicator	Object
`d`	**a directive**
`n`	**a numeral**
`m`	**an erron**
`i`	**an identifer object**
`l`	**a list object**
`a`	**an application object**
`f`	**a function object**

Parsing Messages

star is one of the indicators above.

Message	Explanation
.—	**Message fill**
..—	**Message fill**
...—	**Message fill**
^—	**Error in value quotation**
(—	**Error in parenthesization**
⋆:—	**Error in application's argument**
\—	**Error in formal argument**
\⋆.—	**Error in function body**
[—	**Error before matching '···]'**
<—	**Error before matching '···>'**
{—	**Error before matching '···}'**
!—	**Error after a '!'**
@'='	**Error (nonliteral) before '='**
⋆=—	**Error after an '='**

Chapter 5

Daisy Interpretation

In the Daisy Language Description, evaluation and application are specified as functions

$$\mathcal{V}\colon (E \times \rho) \;\rightarrow\; V$$
$$\mathcal{A}\colon (\rho \times V \times V) \;\rightarrow\; V$$

over expressions E, environments ρ, and values V. Programs `VALUE` and `APPLY` implement functions

$$VALUE\colon (R_E \times R_\rho) \;\rightarrow\; V$$
$$APPLY\colon (R_\rho \times V \times V) \;\rightarrow\; V$$

over *representations*, R_E of expressions and R_ρ of environments, in the space of Daisy values. Where possible, same structures are used as those of the actual Daisy interpreter. However, there are two cases where the actions of interpretation cannot be expressed in Daisy:

> **Daisy programs cannot manipulate errons, or erroneous values. There is no test for their presence, and no operations for composing and decomposing them. `VALUE` represents errons as lists. In addition, only a few of Daisy's primitive operations are addressed in the program, and the rest are implemented by linking to the actual interpreter. Consequently, it is possible for the meta-interpreter to generate "live" errons. In other words, the treatment of errors is synthetic and partial.**

Similarly, the representation of function closures involves information fields that cannnot be accessed through Daisy operations. Closures are also given synthetic representations.

The first parts below detail the representation of Daisy expressions, environments, and closures in the value space. These comprise the base types for the evaluation program that follows. Here is an outline of the program presented in this section

1. details expression representation and illustrates how programs can be built in Daisy.

2. shows how environments are represented and develops functions `EXTEND`, `BINDING`, and `LOOKUP` to manipulate them:

3. deals with function closures, defining functions `isCLOSURE?`, `asCLOSURE`, and `exCLOSURE` to test, compose, and decompose them.

4. deals with (synthetic) errons, defining functions `isERROR?` and `asERROR` to test for and create them.

5. presents an interpreter, built on the previously developed representations. The main programs are

   (a) `VALUE`, an evaluation function is described in Section 5.1. An elementary help function, `VOID?` is also defined.

   (b) `APPLY`, the application function, is described in Section 5.2.

   (c) `LIST`, which distributes `VALUE` over list expressions, is described in Section 5.4.

6. Application is further decomposed into

   (a) `asOPN`, which models Daisy operations including the special primitives `let`, `rec`, `fix`, and `val`. `asOPN` has a number of trivial help functions, including tests `LSTdct?`, `SETdct?`, and `IDYdct?`; and argument validation functions `CHKN` and `CHKNxN`. These are developed in Section 5.3.

   (b) `PROBE`, which describes the application of numerals.

   (c) `LISTFUNCTIONAL`, which, together with functions `HEADs` and `TAILs` describe the construction functional.

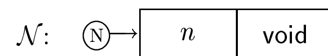
VALUE, APPLY, and LIST are the core interpretation functions. The function asOPN deals with Daisy operations, as well as the special primitives val, let, rec, and fix. PROBE and LISTFUNCTIONAL show the functional interpretation of applied numerals and lists.

5.1 Expression Representation (Figure A)

The Daisy language is a spelling out of expression representations. It is analogous to the *s-expression* notation of Lisp, and these languages differ because of differences in the underlying objects.

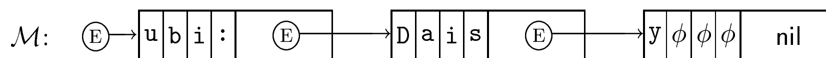
In this section, capitalized italic variables stand for syntactic expressions and caligraphic variables stand for citations to their expression objects: if E is an expression then $\mathcal{E}$ is a reference to the object representing E. The underlying objects are nullary, unary and binary cells, which are discussed further in the Daisy Operations section. Briefly, a *citation* is a reference to an object, together with a tag that classifies that object. The tags indicate both storage classification and an interpretation. A cell is of fixed size, containing binary data and zero, one, or two citations.

A numeral, N, is a unary cell containing N's binary value:



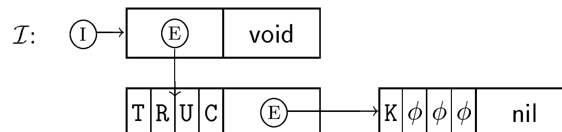
The value n conforms to the host's integer format, a 32-bit two's complement quantity.

An erron is a sequence of unary message cells. The erron |ubi:Daisy| looks like



That is, the binary contents are interpreted as characters, with ϕ being the NUL code.

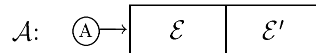
All textual occurances of a literal I, such as TRUCK, resolve to a unique citation to a structure like



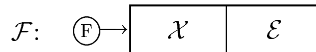
$\mathcal{I}$'s head is a sequence of unary message cells, holding the display name right-filled with ϕs. In isolation, the display name is an erron. $\mathcal{I}$'s tail is an assigned value, if it has one, or the directive void.

Composite expression objects cannot be distinguished by their content. They differ only how they are cited.

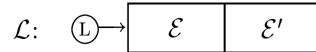
1. **The application expression $E : E'$ is a binary citation:**

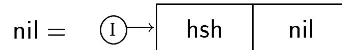
   The function expression $\backslash X\ .\ E$ is a binary citation:

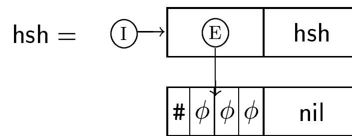
   The pure list expression $[\,E\ !\ E'\,]$ is a binary citation:

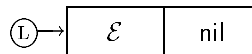
2. **The constant** nil **refers to a unique cell with the form:**

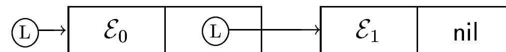
   Nil's head is an identifer object that displays as '#', although this object is distinct from the the literal #**:**

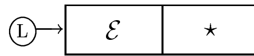
3. **The pure list expression $[\,E\,]$ is equivalent to $[\,E\ !\ [\,]\,]$. Both expressions are incorporated as**

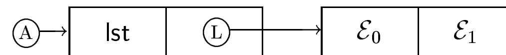
   Both $[\,E_0\ \ E_1\,]$ and $[\,E_0\ !\ [\,E_1\,]\,]$ are incorporated as

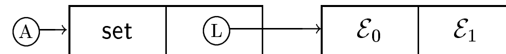
and so on. The expression [E *] reflects a cyclic list object



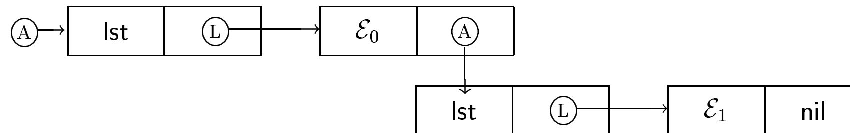
The list expression [E_0 ! E_1] is incorporated as though it had been written lst:[E_0 ! E_1], **where** lst **is a special directive. That is, the form is represented as an application:**



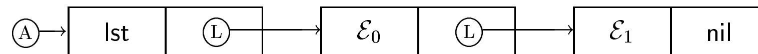
Similarly, a multiset expression set:[E_0 ! E_1] **is incorporated as**



Both [ ] **and** set:[ ] **are represented by** nil, **but angle and brace delimiters do not have a pure dot-notation. The expression** [E_0 ! < E_1 >] **develops the structure**

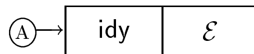


while the expression [E_0 E_1] **is incorporated as.**



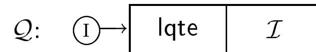
Though these two objects have the same value, interpretation of the latter is faster.

4. **The parenthesized expression (E) is represented as an application:**



The constant idy **is a directive for the identity operation, but is also recognized in parsing and display as indicating parentheses.**

5. **Quotations are represented as identifier objects. In interpretation, they look like assigned literals because they have something other than void in their tails. The literal quotation "$c_0 \cdots c_n$" is incorporated as**



where $\mathcal{I}$ cites the literal spelled '$c_0 \ldots c_n$'. The value quotation ^ E is incorporated as

Parsing and display recognize the directive lqte as indicating that $\mathcal{Q}$'s tail should appear surrounded by double-quotes; and vqte as indicating that $\mathcal{Q}$'s tail should appear preceeded by a caret.

5.1.1 Building expressions

Expressions are built and decomposed with tag coercion operations. The following assignments name the relevant directives, lst, set, lqte, and vqte by extracting them from representative forms. This is done to avoid dependence on internal constants, which are subject to change.

```
     |
     | These assignments name special directives
     |
&LST  = head:asLST:^[anything]
&SET  = head:asLST:^{anything}
&LQTE = head:asLST:^"anything"
&VQTE = head:asLST:^^anything
```

With these constants, the following expressions build expression objects. Assume expressions X, E, E_0, $\ldots$, E_n have values x, e, e_0, $\ldots$, e_n.

Expression	Value
asFTN:[X ! E]	\ x . e
asAPL:[E_0 ! E_1]	$e_0 : e_1$
asIDE:[&IQTE ! I]	"i"
asIDE:[&VQTE ! E]	^e
[$E_0 \cdots E_n$]	[$e_0 \cdots e_n$]
asAPL:[&LST $E_0 \cdots E_n$]	$< e_0 \cdots e_n >$
asAPL:[&SET $E_0 \cdots E_n$]	$< e_0 \cdots e_n >$

The expression `[]` builds Nil; that is, writing '`[]`' is exactly like writing '`[]`'. An value that displays as '`[]`' is gotten by `asAPL:[&LST ! []]`.

An easier way to construct expressions is to use the `xparse` operation, which accepts nonatomic values as neutral symbols. The operation concatenates the incorporated prefix of a symbol stream to the ensuing suffix, so that the head of `xparse`'s result is the desired form. As above, assume the subexpressions on the left return expression objects. The examples below illustrate how to build composite forms (See also the description of `xparse`).

Expression	Value
xparse:["^" E ···]	[^e ···]
xparse:[E_0 ":" E_1 ···]	[$e_0 : e_1$ ···]
xparse:["\\" X "." E ···]	[\ x . e ···]
xparse:["[" E_0 ! E_1 "]" ···]	[[e_0 ! e_1] ···]
xparse:["[" $E_0 \cdots E_n$ "]" ···]	[[$e_0 \cdots e_n$] ···]
xparse:["[" E "*" "]" ···]	[[e *] ···]
xparse:["[" $E_0 \cdots E_n$ "]" ···]	[[$e_0 \cdots e_n$] ···]
xparse:["{" $E_0 \cdots E_n$ "}" ···]	[set:[$e_0 \cdots e_n$] ···]
⋮	⋮

Literal quotations are developed by the `scan` operation, not the

parser. The expression `asIDE:[ &LQTE ! I ]` forms the quotation "I" of the literal I.

5.2 Environment Representation

Recall that environments are modeled as functions from literals to values, extended according to

$$\rho\left[\begin{matrix} v \\ I \end{matrix}\right](J) = \begin{cases} v, & \text{if } I = J \\ \rho(J), & \text{if } I \neq J \end{cases}$$

$$\rho\left[\begin{matrix} [\,u \;!\; v\,] \\ [\,X \;!\; Y\,] \end{matrix}\right](J) = \rho\left[\begin{matrix} v \\ Y \end{matrix}\right]\left[\begin{matrix} v \\ X \end{matrix}\right](J)$$

$$\rho\left[\begin{matrix} v \\ [\,] \end{matrix}\right](J) = \rho(J)$$

This association is represented by a list

$\mathcal{R}$: (L) ⟶ [$\mathcal{F}$ | $\mathcal{A}$]

$\mathcal{F}$ is called the *formal environment;* it contains all bound identifers. $\mathcal{A}$ is the *actual environment;* it contains all identifier bindings. Extending an environment adds separately to both parts. If $\mathcal{R}$, above, represents ρ then $\rho\left[\begin{matrix} v \\ X \end{matrix}\right]$ looks like

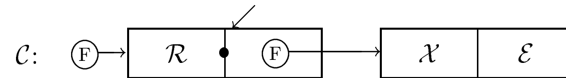
$\mathcal{R}'$ ⟶ [(L) | (L)] ⟶ [$\mathcal{V}$ | $\mathcal{A}$]
first (L) ↓ [$\mathcal{X}$ | $\mathcal{F}$]

The function `EXTEND`, on the next page, builds this structure. The function `BINDING` implements $\rho(I)$, looking up I in the environment structure. `LOOKUP` returns that value in the actual environment which is in the same position as `I` in the formal environment. If `I` does not occur in `F`, the value `V` is returned. `BINDING` sets `V` to be an erron, reporting that the sought identifier is unbound. For instance, if the literal `WRONG` is unbound, then `BINDING` returns `|ubi:WRONG|`. The function `asERROR`, which models erron creation, is defined later in Section 4. The search is `head`-first; that is, an occurance of `I` in `F`'s head takes precedence over an occurance in `F`'s tail. The outcome is erroneous if the formal argument contains something other than lists and literals. See also, Techinal Note ?.2–1.

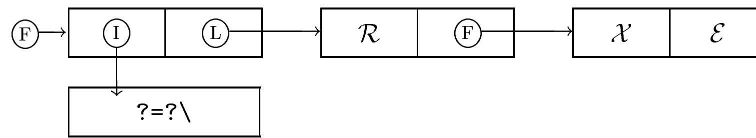
```
   |
   | EXTEND:[Formal, Value, Environment] --] Environment
   | Environment: [Formals ! Actuals]
   |
EXTEND = \[X V R] .
 let:[ [F ! A]
       R
       [ [X ! F] ! [V ! A] ]
      ]

  |
  | BINDING:[Literal, Environment] --] Value
  |
BINDING = \[I R]. LOOKUP:[I R asERROR:["ubi:" I] ]

  |
  | LOOKUP:[Literal, Environment, Value] --] Value
  | Environment: [Formals ! Actuals]
  |
LOOKUP = \[ I  [F ! A]  V ] .
  if:[ nil?: F
          V

       isLtrl?:F
          if:[ same?:[I F]  A  V ]

       isLST?:F
          LOOKUP:[ I
                   [head:F ! head:A]
                   LOOKUP:[ I
                            [tail:F ! tail:A]
                            V
                  ]        ]

       "otherwise"
          asERROR:["arg/" I]
     ]
```

5.3 Closure Objects

The closure object, denoted $_\rho\backslash X\,.\,E$ in the Daisy Language Definition, is represented as a function cell



The closure records the environment in effect at the point of function's evaluation; $\mathcal{C}$'s tail is just the original function object, $\backslash X\,.\,E$. Closures are distiguished from functions by a mark (•) that is inaccessible to Daisy programs. Hence, the implementation of closures artificially marks them by attaching a literal indicator '?=?\'. This choice of indicator mimics what is seen when closures are displayed. The synthetic closure object is



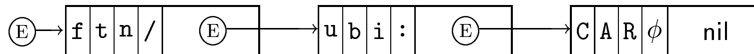
Closures are built by `asCLOSURE`**. The test** `isCLOSURE?` **determines whether a function object is closed. The destructuring function** `exCLOSURE` **returns the content of a closure object, considered as a list [See Technical Note ??3–1].**

```
     |
     | &CLOSURE is the reserved indicator for closure objects
     |
&CLOSURE = "?=?\"

     |
     | asCLOSURE:[Environment, Function] --] Closure
     |    (closes a function object)
     |
asCLOSURE = \[R F].  asFTN:[ &CLOSURE R ! F ]

     |
     | isCLOSURE?:FunctionObject --] Bool
     |    (test for a closure object)
     |
isCLOSURE? = \F.  same?:[ &CLOSURE  head:asLST:F ]
```

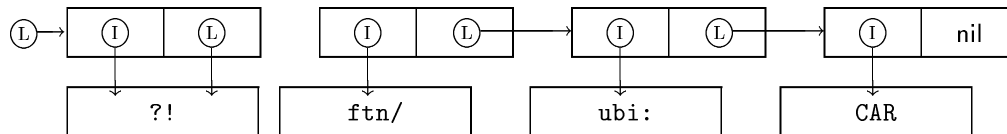
```
      |
      | exCLOSURE:ClosureObject --] [Envt., FormalArg. ! Body]
      |     (exposes the components of a closure object)
      |
   exCLOSURE = \C.
     let:[ [ Mark R ! F ]  asLST:C
           [ R ! asLST:F ]
         ]
```

5.4 Errons

Daisy programs can result in errons, but cannot test for them or manipulate them. Hence, the treatment of errors cannot be emmulated with the "live" objects [See Technical Note ??4–1]. The functions below maintain a representation of errors as lists of literals, headed with indicator '`?!`'. The erron `|ftn/ubi:CAR|`, which really looks like



is modeled as




```
     |
     | &ERROR is the reserved indicator for error objects.
     | Erron: [&ERROR Literal, ...]
     |
   &ERROR = "?!"
     |
     | isERROR?: Value --] Bool
     |    (test for a synthetic erron)
     |
   isERROR? = \V. all?:[ isLST?:V
                          same?:[ &ERROR head:V ]
                        ]
```

The function `asERROR` incorporates new text in compound errons (See the Daisy Errons section). Here are some examples:

Expression	Value	Models
asERROR:["arg/"]	[?! arg/]	|arg/|
asERROR:["arg/" asERROR:["ubi:" "CAR"]]	[?! arg/ ubi: car]	|arg/ubi:CAR|

```
    |
    | asERROR: [Literal, ...] --] Erron
    |        : [Literal ! Erron] --] Erron
    |
  asERROR = \[I ! Is].
    let:[ [Mark ! E]
          Is
          [ &ERROR I ! if:[ isERROR?:Is
                             tail:Is
                             []
                          ]
           ]
         ]
```

5.5 A Daisy Interpreter

The two main interpretation programs, `VALUE` and `APPLY` correspond to the evaluation functions $\mathcal{V}$ and $\mathcal{A}$ the Daisy Langage Definition.

5.5.1 Evaluation

`VALUE`
is shown on the next page, with discussion notes below. [1]

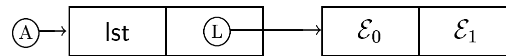
Identifiers `E0` and `E1` name expression `E`'s head and tail, and are used throughout `VALUE`'s body when `E` is a composite structure. See also Technical Note ??5–1. [2] **Where expression `E` is, itself,**

an erron, it is often the byproduct of a syntax error. The parsing operations generate errons to report bad syntax, and these are passed on to evaluation. The interpreter adds the prefix `val/` to the error message. [3] **Directives evaluate to themselves. Typi-**

```
   |
   | VALUE: [Expression, Environment] --] Value
   |
VALUE = \[E R] .                | NOTES
                                 |
 let:[ [E0 ! E1]                 |  [1]
       asLST:E                   |
                                 |
  if:[ isERROR?:E                |  [2]
          asERROR:["val/" ! E]   |
       isDCT?:E                  |  [3]
          E                      |
       isNML?:E                  |  [4]
          E                      |
       isLST?:E                  |  [5]
          E                      |
       isFTN?:E                  |  [6]
          asCLOSURE:[ R  E ]     |
       isIDE?:E                  |  [7]
          if:[ VOID?:E1    |
                  BINDING:[E R]  |
                  E1             |
              ]                  |
       isAPL?:E                  |  [8]
         (( APPLY:R              |
          ): VALUE:[E0 R]        |
          ): VALUE:[E1 R]        |
                                 |
     ]]                          |  [9]
                                 |
```

cally, there are just three cases where evaluation encounters directives, rather than their names, in applications. The lst, set, and idy directives are incorporated in parsing. See Section 1, and also Note [5], below. [4] Numerals evaluate to themselves. [5] Pure

list objects evaluate to themselves. The typical list expression, [E_0 ! E_1] is incorporated as



which applies the value of lst, a directive, to the value of [E_0 ! E_1]; both evaluations are immediate. In addition, the actual interpreter specialized for this case. Applications of binding primitives, like `let:[` X E_0 E_1 `]` use pure-list brackets to supress evaluation of their arguments. [See Technical Note ??5–2.] [6] Evalua-

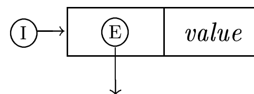
tion of a function produces a closure. There is *no* test at this point to determine if `E` is already closed. An expression like `val:val:^\X.E` produces a twice-closed function object. Should this object be applied, the result is useless. [7] This point in

`VALUE` accounts for Nil, literals, and quotations. The `VOID?` test is defined as

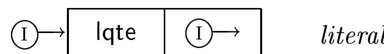
```
     |
     | VOID? tests for the |void| constant.
     |
  VOID? = \V. same?:[V asDCT:0]
```

The void value `asDCT:0` cannot be assigned to a symbolic name, for to do so has the effect of *un*assigning the name. Where `E1` is void, `E`'s binding is sought in the environment. Should `E` be unbound, the `BINDING` function returns an erron.

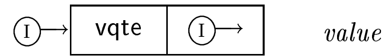
The test `VOID?:E1` fails when `E` is an assigned literal



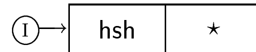
a literal quotation



a value quotation



or Nil.



In each case `E1` is the desired value. [8] Interpretation of an

application object applies the value of the function-part to the value of the argument-part. The curried form of `APPLY`, invoked as

$$\texttt{((APPLY:R):}\ \mathcal{V}_0\texttt{):}\ \ \mathcal{V}_1 \quad ,$$

enforces Daisy's strictness in application. That is,

$$\texttt{((APPLY:R):}\ \Omega\texttt{):}\ \ \texttt{E1} = \texttt{((APPLY:R): E2):}\ \ \Omega = \bot$$

The order of evaluation is accurate: `E0` then `E1`, both before control transfers to `APPLY`. [9] The cases are exhaustive because there

are just seven kinds of object in the data space. The sequential behavior of the actual interpreter is better described as a case-statement, or jump table, indexed by the citation `E`'s tag. The behavior is *almost* described by

```
(TagOf:E): [ erron-case
             directive-case
             numeral-case
             list-case
             function-case
             identifier-case
             application-case
           ]   ]
```

With the synthetic representation, errors would be handled in *list-case*; at the implementation level, there is a distinct tag for errons. The actual numeric values of tags is not correct in the expression above—consult the description of `TagOf` for the true order.

5.5.2 Application

As in VALUE, the APPLY function is a selection based on the applied value's tag. The coding of APPLY is shown on the next page, with notes just below. The functional interpretation of directives, numerals, and lists are developed later in asOPN, PROBE, and LISTFUNCTIONAL. Application of identifer and application objects is erroneous, but see Note [5]. [1] APPLY has the form (\R.\U.\V. ···) to enforce Daisy's strictness in application. See VALUE's Note [8]. Thus,

$$\texttt{APPLY}\,\rho\ \perp\ v\ =\ \texttt{APPLY}\,\rho\, u\ \perp\ = \perp$$

as discussed in the language description. [2] If the function-value is erroneous, APPLY prefixes ftn/. For instance, if BAD is unbound, the expression BAD:9 yields |ftn/ubi:BAD|. [3] Application of a directive is decoded by asOPN, which returns a function expecting an environment and an argument. The environment is passed for operations val, let, rec, and fix; and also for the lst and set primitives (See VALUE's Note [3]. All other operations, ignore the argument R. [4] The PROBE function returns the U$^{\text{th}}$ element of V; its description is developed later. [5] Nil is the only identifier object with a functional interpretation, and it is the constant-Nil function. This interpretation fits that of the construction functional. The erron generated in this case includes the identifier's name, if it is a literal. For instance, the expression "BAD":9 returns |ftn/BAD|. Compare this with Note [3], above. [6] No application object has a functional interpretion. Such objects are unlikely to arise at this point, except through incorrect program construction. [7] Daisy's construction functional is developed later as the function LISTFUNCTIONAL. [8] Function objects represent both closures and function expressions, and the isCLOSURE? tests discriminates between them.

- **At this point, according to closure representations,**

X	**is a formal argument**
R'	**is an environment**
E	**is an expression**

```
   |
   | APPLY: Envt. --] Value --] Value --] Value
   |
                                    | NOTES
APPLY = \R. \U. \V .               | [1]
                                    |
  if:[ isERROR?:U                   | [2]
          asERROR:["ftn/" ! U]      |
       isDCT?:U                     | [3]
         (asOPN:U):[R V]            |
       isNML?:U                     | [4]
         PROBE:[U V]                |
                                    |
       isIDE?:U                     | [5]
         if:[ nil?:U                |
                 []                 |
                 asERROR:["apl/"]   |
             ]                      |
                                    |
       isAPL?:U                     | [6]
         asERROR:["apl/"]           |
       isLST?:U                     | [7]
         LISTFUNCTIONAL:[R U V]     |
                                    |
       isFTN?:U                     | [8]
         if:[ isCLOSURE?:U          |
                                    |
                 let:[ [ R' X ! E ] | [8a]
                       exCLOSURE:U
                       VALUE:[ E EXTEND:[X V R'] ]
                      ]
                                    |
                 let:[ [X ! E]      | [8b]
                       asLST:U
                       VALUE:[ E [X ! V] ]
                      ]
             ]                      |
    ]                               | [9]
                                    |
```

If U is a closure, its environment, formal argument, and body are retrieved; the closure's environment is extended; and the body is evaluated.

- At this point, according to environment representations,

X	is a formal argument
V	is a value
[X ! V]	is an environment
E	is an expression

If U is a function expression, it is interpreted as a combinator, requiring no bindings beyond the pairing of the formal and actual arguments. Thus, any free occurance within the body becomes unbound.[9]The cases are exhaustive because there

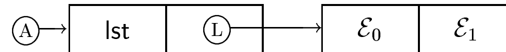
are just seven kinds of object in the data space. See VALUE's Note [9].

5.5.3 Operations

The asOPN function, shown on the next page, converts the directive encoding an operation into a function modeling that operation. The program is only partly defined to illustrate a few key operations; asOPN invokes the underlying primitives for those directives not described. For this reason, Daisy's treatment of errors is not fully described. This point is explained more fully in the notes.

asOPN takes a directive and returns a function expecting an environment, R and a value. R is ignored by all but the special operations, such as let and val.

[1] Recall that an expression like [E_1 ! E_2] is incorporated as an application,



where lst is the list-directive. Evaluation of this form applies lst to pure-list argument, [E_1 ! E_2] (See VALUE's Note [5]). LSTdct?, and SETdct? in the next case, test for the two directives. There is just one other directive explicitly incorporated in applications by the parser: it is the identity operation idy, associated with a

```
   |
   | asOPN: Directive --] ( [Envt., Value] --] Value )
   |
asOPN = \D .                                     | NOTES
                                                 |
 if:[ LSTdct?:D                                  | [1]
        \[R Es]. LIST:[R Es "D"]                 | [2]
      SETdct?:D                                  | [1]
        \[R Es]. LIST:[R Es "I"]                 | [3]
      same?:[D val]                              | [4, 5]
        \[R E]. VALUE:[E R]                      |
                                                 |
      same?:[D let]                              | [4, 6]
        \[R [X E E']] .                          |
            VALUE:[ E'                           |
                    EXTEND:[ X VALUE:[E R] R ]|
                   ]                             |
                                                 |
      same?:[D rec]                              | [4, 7]
        \[R [X E E']] .                          |
            rec:[ R'                             |
                  EXTEND:[ X                     |
                           VALUE:[E R']          |
                           R                     |
                          ]                      |
                  VALUE:[E' R']                  |
                 ]                               |
                                                 |
      same?:[D fix]                              | [4, 8]
        \[R [X ! E]] .                           |
             fix:[ V                             |
                 ! VALUE:[ E                     |
                           EXTEND:[ X V R ]      |
                          ]                      |
                  ]                              |
                                                 |
```

parenthesized expression (See Section 1). Here, these tests are implemented by

```
      |
      | LSTdct?: Directive --] Bool
      | SETdct?: Directive --] Bool
      |
   LSTdct? = let:[ X
                   head:asLST:^[anything]
                   \D. same?:[D X]
                  ]
```

```
   SETdct? = let:[ X
                   head:asLST:^{anything}
                   \D. same?:[D X]
                  ]
```

This way of defining the tests makes them independent of possible changes to the actual constants. A similar thing is done in Section 1.1. [2] **The** lst **directive transfers control to the** `LIST` **function, de-**

scribed later. Briefly, `LIST` **maps** `VALUE` **over a list-of-expressions. The argument** `B` **is a binary indicator, telling** `LIST` **which constructor to use. In this case,** `"D"` **indicates the determinate constructor.** [3] **The** set **directive also transfers control to** `LIST`**, the only differ-**

ence being the indicator `"I"`**, telling** `LIST` **to use the indeterminate constructor.** [4] **The test** `same?:[D val]` **compares the directive**

`D` **with** `val`**'s value, which is the directive encoding the evaluation operation. Similar tests are done throughout** `asOPN`**. In the actual interpreter, invokation of operations is done through a jump table, indexed by** `D`**. A more accurate description of** `asOPN` **would**

be

```
asOPN = \D.                              |
    let:[ UnknownOperation               |
          \[R V]. asERROR:["opn/"]       |
                                         |
         (asNML:D):                      |
            [ UnknownOperation           |  0
              UnknownOperation           |  1
              UnknownOperation           |  2
              \[R E]. VALUE:[E R]        |  3 = val
              \[R Es]. Es                |  4 = idy
              \[R Es]. LIST:[R Es "D"]   |  5 = lst
              \[R Es]. LIST:[R Es "I"]   |  6 = set
              UnknownOperation           |  7
              \[R [X E E']].···          |  8 = let
              \[R [X E E']].···          |  9 = rec
              \[R [X ! E]].···           | 10 = fix
              ...
             ]                           | and so on
```

The numeric value of an operation's directive is subject to change. [5] The val operation invokes VALUE, using the environment in effect where val is applied. In that sense, val is dynamically scoped. For instance, the following expression returns 5, even though F is bound to val in an environment binding X to 0:

```
let:[ X 0
let:[ F val
let:[ X 5
      F:"X"
     ]]]
```

The other binding operations have the same property. [6] The let operation expects a list, consisting of a formal argument X, a binding expression E, and a body E'. The body is evaluated in an environment that extends R by E's value in R. R is the environment in effect where let is applied. The expression

```
let:[ X 5
let:[ F let
let:[ X 6
      F:[Y X Y]
     ]]]
```

returns 6, even though F is bound at a point where X is 5.

EXTEND uses a lazy constructor to build the new environment. As a consequence, an expression like let:[X Ω E] does not diverge unless evaluation of E calls for a binding from X. The "macro expansion" to (\ X . E):Ω would diverge because application is strict. [7] The rec operation expects a list, consisting

of a formal argument X, a binding expression E, and a body E'. The rec operation fixes a new environment R' for E and E', making X's new binding available for both evaluations. This is implemented, in the usual way, by pre-allocating the object for R', and later updating its content. [8] The fix operation expects a list,

whose head is a formal argument and whose tail is an expression. fix:[X ! E] is like rec:\LST{X\ E\ E}. In describing fix, it is more natural to fix the value, V. However, the implementation uses the same mechanism as rec's to build a recursive environment (Note [7] above), then simply retrieves the initial binding. Recall that EXTEND represents an environment as a structure

```
[[X ! F] ! [V ! A]]
A more accurate portrayal of |fix| is
\CenterCode
rec:[ R'
      EXTEND:[ X VALUE:[E R'] R ]
      head:tail:R'
     ]
```

[9] The remaining operations discard the environment R and are,

therefore, true operations and not extensions of the language. Only three, add, inc, and dcr, are described, in order to indicate the error treatment. Unary arithmetic operations, like inc and dcr, expect a numeral and share a run-time validation,

```
    |
    | CHKN:[Value, (Nml --] Value)] --] Value
    |
CHKN = \[V O].                                  | NOTES
  if:[ isNML?:V                                  |
         O:V                                     | [9.1]
         asERROR:["nnO/"]                        |
      ]                                          |
```

Similarly, binary arithmetic operations, like `add`, share the argument validation `CHKNxN`.

```
    |
    | CHKNxN?:[Value, ([Numeral, Numeral] --] Value)] --] Value
    |
  CHKNxN? = \[V O].                              | NOTES
     if:[ all?:[ isLST?:V                        | [9.2, 9.3]
                 isNML?:head:V                   |
                 isLST?:tail:V                   | [9.2]
                 isNML?:head:tail:V ]            |
            O:V                                  | [9.1]
            asERROR:["nn?/"]                     | [9.3]
        ]                                        |
```

[9.1]`CHKN` **and** `CHKNxN` **invoke the pending operation** `O` **provided** `V` **is**

a numeral, in the first case, or a pair on numerals in the second case. Otherwise, an erron is generated. The erron does not record which operation was to be invoked. [9.2]**In the implementation,**

the `asLST?` **tests are implicit to the access primitives,** `head` **and** `tail`**. This cannot be accurately described at this level. In fact,** `add`**'s argument can be any binary object, such as a function cell.** [9.3]**In** `CHKNxN` **there is no verification of the argument's length.**

The expression `add:[2 3 4]` **returns the sum of** 2 **and** 3. [9.4]

If `V` **does not conform to the expected structure, and erron is created. It can be either** `|nn0/···|` **or** `|nn1/···|`**, depending on which test fails (See the Daisy Errons section. Implementations of** `CHKNxN` **and** `CHKN` **share code in a manner not described here.** [10] **Detailed descriptions of the remaining Daisy operations are**

omitted (See the Daisy Operations section). Instead, the default case simply applies the directive to the argument, thereby linking to the actual interpreter. In doing this, description is no longer insulated from true errors—`isERROR?` **doesn't work for *real* errons.**

5.5.4 List Evaluation

The function LIST maps VALUE over an expression list.

```
      |
      | LIST: [Envt, [Expression, ...] B] --] [Value, ...]
      |    where
      |    B: {"D", "I"}
      |
   LIST = \[R L B].                                    | NOTES
     let:[ [E ! L']  L                                  |
     let:[ V   VALUE:[R E]                              |
     let:[ Vs  LIST:[R L']                              |
                                                        |
      if:[ isLST?:L                                     |
             if:[ same?:[L L']                          | [1]
                    [V *]                               |
                  same?:[B "D"]                         | [2]
                    [V ! Vz]                            |
                  same?:[B "I"]                         |
                    {V ! Vz}                            |
                  ]                                     |
           "otherwise"                                  |
             VALUE:[R L]                                | [3]
          ]]]]                                          |
```

[1] A cyclic expression list results in a cyclic value list. [2] Where

L is not a cyclic list, LIST iterates. The binary indicator B tells which data constructor to use, D for "determinate" and I for "inderterminate". [3] LIST degenerates to VALUE if L is not a list.

In particular, this branch takes care of Nil.

In implementation, LIST is more elaborite. The following points are not modeled by the program above.

- There is an explicit check for Nil.
- Expressions E are partially evaluated. For instance, should E be a directive, numeral, or list, it is not evaluated; if it is a function, E is closed immediately. An erron encountered during partial evaluation is simply prefixed with |lst/ $\cdots$ |.
- Since partial evaluation tests E, LIST can diverge if E is $\perp$ (*not* Ω), that is, if the expression's *construction* diverges.

- **Similarly, `LIST`'s iteration is more strict than is described by `[V ! Vs]`. Evaluating `[`E `!` $\perp$`]` (*not* `[`E `!` Ω`]`) can diverge.**

In typical cases, `LIST`'s strictness is justified by the presumption that programs are manifest when they execute. This is true where Daisy's parser is involved because it too is strict. For this reason, explicit use of the lst directive is not recommended. The `evlst` operation more safely maps evaluation.

5.5.5 Numeric Probes

The `PROBE` function indexes the (list) value `L` according to the numeral `N`.

```
    |
    | PROBE: [Numeral Value] --] Value
    |                                              | NOTES
 PROBE = \[N L].                                  |
   rec:[ Index                                     | [2]
         \[N [V ! Vs]].                            |
             if:[ zero?:N                          | [3]
                    V                              |
                  isLST?:Vs                        | [4]
                    Index:[dcr:N Vs]               |
                    asERROR:["prb/" ! Vs]          |
                 ]                                 |
                                                   |
         if:[ all?:[ pos?:N  isLST?:L]             | [1]
                Index:[N L]                        |
                asERROR:["prb/" ! L]               |
             ]                                     |
        ]                                          |
```

`[1]` Indexing proceeds once it is verified that `N` is non-negative and that `L` is a list. Otherwise an erron is generated, with prefix `|prb/`$\cdots$`|`. In the implementation, the `pos?` test actually checks whether `N` is between zero and the largest list address. `[2]` `PROBE` is an iterative loop, called `Index` here. `[3]` `Index` invokes `N` `tails`, followed by a `head`. That is, indexing starts with zero. `[4]` A probe error results if something other than a list is encountered, such as Nil.

5.5.6 The Construction Functional

An applied list, call it $\mathcal{L}$, is interpreted as a mapping functional, The argument should be a list of lists, which, for the moment is thought of as a "row major" matix, $\mathcal{M}$. $\mathcal{L}$'s head is applied to $\mathcal{M}$'s first column, and $\mathcal{L}$'s tail is applied to the matix resulting when $\mathcal{M}$'s first column is removed.

Where $\mathcal{L}$ is a proper list—no funny tails—and $\mathcal{M}$ is rectangular, the construction functional can be thought of as an element-wise application of $\mathcal{L}$'s elements to $\mathcal{M}$ *transposed*. There are several boundary conditions, also descibed in the Daisy Langugae Description:

- $\mathcal{M}$ can be nonfinite in either dimension.
- In effect, $\mathcal{M}$'s rows are padded with values appearing as '#'.
- Where $\mathcal{L}$ is cyclic, there is an extra termination check; the result has the length of $\mathcal{M}$'s first row.

The construction functional is described by the LISTFUNCTIONAL program, below, with notes following.

```
   |
   | LISTFUNCTIONAL: [ Envt List Value] --] List
   |
LISTFUNCTIONAL = \[ R F V ].                    | NOTES
   let:[ [ Fh ! Ft ] F                           | [1]
   let:[ APPLY' APPLY:[[]![]]                    | [2]
                                                 |
    if:[ nil?:isLST?:V                           | [3]
            asERROR:["f-c"]                      |
                                                 |
         all?:[ same?:[F Ft] nil?:head:V ]       | [4]
            []                                   |
                                                 |
         [ (APPLY':Fh):HEADs:V                   | [5]
         ! (APPLY':Ft):TAILs:V                   |
          ]                                      |
        ] ]]                                     |
                                                 |
```

[1] The argument F is known to be a list at this point. [2] The

environment at the point of application is discarded. The expression `[[] ! []]` **forms an environment with empty formal and actual parts (in the implementation the environment is simply set to Nil). Dereferencing the environment was done to repair space consumption problems in certain programs. Here is a pathelogical example of why this might be considered wrong: the expression**

```
let:[ X  5
      [let *]:[ ["Y"  "Y" ]
                [ X   "X" ]
                ["Y"  "Y" ]
              ]
     ]
```

returns `[5 |ubi:X|]`**, while its expansion,**

```
let:[ X  5
      [ let:["Y"  X "Y"]
        let:["Y" "X" "Y"]
       ]
     ]
```

returns `[5 5]`**. It is not clear what these kinds of programs** ***should*** **mean, but they have occasionally arisen in applications modeling interactive environment. In this case, one can code around the problem by replacing** `[let *]` **with** `[(\v.let:v) *]`**. [3] The**

argument, `V` **is verified as a list, but it is not verified that** `V`**'s elements are lists—as they should be. [4] The test** `same?:[F Ft]`

succeeds if `F` **is a cycle, and if so, construction terminates when** `V`**'s first row is Nil. Though it is not described here, the termination test generates the** `|f-c|` **erron when** `L`**'s head is neither a list nor Nil. [5] The remaining case performs the mapping, with** `HEAD`**s**

and `TAIL`**s incrementally transposing the argument. The functions below do not illustrate the all the checking, which is discussed in**

the commentary.

```
      |
      | HEADs: List --] [List ...]
      | TAILs: List --] [List ...]
      |-------------------------------------------| NOTES
                                                  |
   HEADs = \L.                                    |
     let:[ [[H ! T] ! L' ]                        | [5.1]
           L                                      |
           [ H                                    |
           ! if:[ nil?:L'                         | [5.2]
                    []                            |
                  isLST?:L'                       |
                    HEADs:L'                      |
                    asERROR:["xps"]               |
                ]                                 |
            ]                                     |
          ]                                       |

                                                  |
   TAILs = \L.                                    |
     let:[ [[H ! T] ! L' ]                        | [5.1]
           L                                      |
           [ T                                    |
           ! if:[ nil?:L'                         | [5.2]
                    []                            |
                  isLST?:L'                       |
                    TAILs:L'                      |
                    asERROR:["xps"]               |
                ]                                 |
            ]                                     |
          ]                                       |
                                                  |
```

`[5.1]`**There is an additional test on `L`'s head to verify that it is a list. If not, the `|xps|` erron is returned.**`[5.2]`**The test on `L`'s tail, `L'`, is actually made prior to construction. Hence, `HEADs` and `TAILs` are more strict in implementation than is described here. (See also `LISTFUNCTIONAL`'s Note `[2]`). This has lead to difficulty in programming certain applications. In practical experience, the conditions leading to problems are manipulation of interactive input**

to which cyclic lists are applied; and the symptom is a premature input prompt. Since there are other—equally obscure—reasons this symptom can arise, correcting such programs can be hard. A trick that almost always works is to replace an application of [F *] with a value Fs, defined to be

```
Fs = fix:[ L ! [F F ! L] ]
```

Since Fs is a cycle of length two, the cyclic-termination test is not made. (See also, Technical Note ??5–3)

5.5.7 Testing the Evaluation Model

The program TEST, below, interactively exercises the system of functions described in this section. TEST's single argument, a literal, is a terminal prompt.

```
TEST = \Prompt .
   let:[ InitialEnvironment
         [ NmlAsChr:10 ! NmlAsChr:10 ]

   let:[ Operator
         xparses:scans:console:Prompt

         [VALUE *]:[ Operator
                     [ InitialEnvironment *]
                   ]
        ]]
```

InitialEnvironment binds the new-line character NL to itself; NmlAsChr:10 gives the literal character with display code 10. This binding is needed because parsing passes new-lines to evaluation.

VALUE is mapped over a stream of expressions, using the initial environment with each. The expression console:Prompt establishes a channel to the operator's keyboard with the given prompt, producing a stream of characters corresponding the operator's key strokes. Given such a stream, scans builds a stream of atoms: literals, numerals, and symbols. Given a stream of atoms, xparses produces a stream of represented expressions. It is this stream, called Operator, to which VALUE is applied element-wise.

TEST shows that VALUE is interpreting the same expression representations as are used by the actual interpreter. It also specifies

how interactive Daisy is established in its implementation. However, a closer description is given by replacing TEST's body with

```
VALUE:[ asAPL:[ &LSTdct ! Operator ]
        InitialEnvironment
       ]
```

where &LSTdct is the lst directive. That is, VALUE is given a single expression representing [*inputexpressions*]. In the implementation the evlst directive is used rather than lst (See the discussion of LIST).

The program shown on the next page is a version of the system described earlier, which does not use assignment. An outline of the program is

```
INTERPRET = \Z.
let:[ | Constants
      [ &CLOSURE &ERROR &LSTdct &SETdct ]
      [ ··· values ··· ]
let:[ | Combinators
      [ EXTEND asCLOSURE isCLOSURE? exCLOSURE isERROR? asERROR
        VOID? LSTdct? SETdct? CODES? CHKN CHKNxN?
       ]
      [ ··· values ··· ]
let:[ | Trivial Loops
      [BINDING PROBE HEADs TAILs]
      [ ··· values ··· ]
rec:[ | Serious loops
      [ VALUE APPLY asOPN LIST LISTFUNCTIONAL ]
      [ ··· values ··· ]
      | in
        VALUE:Z
      ]]]]
```

Given an expression E, the test program below invokes INTERPRET

on itself, interpreting E.

```
TESTTEST = \Expression.
   let:[ InitialEnvironment
         [[]![]]
         INTERPRET:[ asAPL:[ INTERPRET
                           ! [Expression InitialEnvironment]
                            ]
                     InitialEnvironment
                   ]
        ]
```

This is an intensive trial, but it is not exhaustive because the construction functional is not used in INTERPRET.

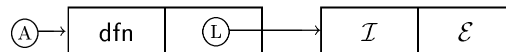
5.6 Assignment

Though assignment is to be considered outside the Daisy language, the command is implemented by Daisy's interpreter. It is discussed here for completeness. However, the means of assignment, and the privilege to invoke it, are in no way guaranteed in future versions of the language.

The parsing operations test for the = symbol in their outer loops. Where the form

Literal = *Expression*

is recognized, the following representation is built.



That is, the command is incorporated as an appliaction of the dfn directive (meaning "define").

Where such a form is evaluated, dfn and the list argument are passed to APPLY, and subsequently to asOPN. In other words, the assignment command has the same evaluation pattern as a [···] expression.

In asOPN, the case for dfn is something like

```
DFNdct?:D
  \[R [I ! E]].  (ASSIGN:I): VALUE:[E R]
```

The DEFINE function is curried to specify that E is evaluated prior to the assignment. Functionally, DEFINE simply returns I, the name assigned, and not the assigned value.

```
      |
      | ASSIGN: Literal --] Value --] Literal
      |    Includes a side effect.
      |
   DEFINE = \I. \V. set:[  ]I
```

5.7 Technical Notes

NOTE 2–1.

In LOOKUP, it is more accurate to use the access operations _hd and _tl in place of head and tail in probing the actual environment A. The former do not verify their operands as lists. and one benefit is that function parameters Though this is a little faster, it can lead to mysterious results in wrong programs. Use of head and tail is under consideration for future releases.

NOTE 3–1.

This note is in the category of "telling the *whole truth* about Daisy." It happens that in the 4 release, one can express a closure object. I typed the expression `asFTN:{ ^[F ! A] ! ^\X.E }` , which evaluates to `\?=?\X.E` . This is the issue operation's display of a properly represented closure. This works because:

- The mark that distiguishes function objects from closure objects is the same mark (actually a synchronization bit) that distiguishes determinate lists from indeterminate lists.
- The value returned is not cited as a list and, therefore, is never subject to re-ordering.

These are purely coincidental points about representation. Their exploitation is not only risky, but also depends on knowledge of how multisets are implemented. Such knowledge is beneath the level of description for the language manual. In addition, there is still no means available to test for these objects [but see NOTE 4–1].

NOTE 4–1.

Again, the whole truth. Since print names are indistinguishible from errons—in fact, they are errons—it is possible to create and manipulate them indirectly as literals. One could contemplate an approach to error analysis by passing values through the `issue` **operation and inspecting the resulting character stream. Look for the** | **character.**

NOTE 5–1.

The coercion `asLST:E` **is not really needed because actual-environment probes are not sensitive to lists (See Technical Note 2–1). That is, it would be correct to write** `let:[ [E0 ! E1] E ··· ]` **.**

NOTE 5–2.

Revision to the interpreter—hence also the language, invoking list evaluation at this point, is under consideration. In `VALUE`**, the fragment**

```
isLST?:E                    |  [5]
   E                        |
```

changes, to become

```
isLST?:E                    |  [5]
   LIST:[R Es "D"]    |
```

Should this be done, a different mechanism for applying binding operations (i.e., `let`**,** `rec`**, etc.) is needed.**

NOTE 5–3.

The early test in `HEAD`**s and** `TAIL`**s are implemented to eliminate backward references in stream computations. Let S = [$\mathcal{N}$, ···] and T = [$\mathcal{M}$, ···] be nonterminating integer lists. Then the application of** [add ∗] **to** [$\mathcal{S}$ $\mathcal{T}$] **maps** add **over the stream** [[$\mathcal{N}$, $\mathcal{M}$,] ···]**. Now, each pair in this list is verified by** `CHKNxN`**, which never tests the pairs' lengths (See** `asOPN`**'s Note** `[9.3]`**). Unless the early test is made, trivial but unfinished invokations of** `HEAD`**s and** `TAIL`**s retain a chain of backward references in the stream. As a consequence, programs that should run in bounded space do not.**

```
 same?:[D inc]                             | [4, 9]
   \[R V]. CHKN:[ V add ]                  |
 same?:[D dcr]                             | [4, 9]
   \[R V]. CHKN:[ V mpy ]                  |
 same?:[D add]                             | [4, 9]
   \[R V]. CHKNxN?:[ V inc:V ]             |
                                           |
"otherwise"                                | [10]
   \[R V] . D:X                            |
 ]                                         |
                                           |
```

```
INTERPRET = \Z.
let:[ | Constants
      [ &CLOSURE &ERROR &LSTdct &SETdct ]
      [ "?=?\"    | = &CLOSURE
        "?!"    | = &ERROR
        head:asLST:^[anything]  | = &LSTdct
        head:asLST:^{anything}  | = &SETdct
       ]
let:[ | Combinators
      [ EXTEND asCLOSURE isCLOSURE? exCLOSURE isERROR? asERROR
        VOID? LSTdct? SETdct? CODES? CHKN CHKNxN?
       ]
      [ | EXTEND =
          \[X V R] . let:[ [F ! A] R [ [X ! F] ! [V ! A] ] ]
        | asCLOSURE =
          \[R F].  asFTN:[ &CLOSURE R ! F ]
        | isCLOSURE? =
          \F.  same?:[ &CLOSURE  head:asLST:F ]
        | exCLOSURE =
          \C.  let:[ [ Mark R ! F ]  asLST:C [ R ! asLST:F ] ]
        | isERROR? =
          \V. all?:[ isLST?:V same?:[ &ERROR head:V ] ]
        | asERROR =
          \[I ! Is].  let:[ [Mark ! E] Is
                            [ &ERROR I ! if:[ nil?:Is
                                                  []
                                                same?:[&ERROR Mark]
                                                  Es
                                                  Is
                                               ]
                             ]
                            ]
        | VOID? =
          \V. same?:[V asDCT:0]
        | LSTdct? =
          \D. same?:[D &LSTdct]
        | SETdct? =
          \D. same?:[D &SETdct]
        | CODES? =
          \[D I]. same?:[D tail:asLST:I]
        | CHKN =
          \[V O].  if:[ isNML?:V O:V "nnO" ]
        | CHKNxN? =
          \[V O]. if:[ all?:[ isLST?:V isNML?:head:V
                              isLST?:tail:V isNML?:head:tail:V ]
                       O:V
                       "nn?"
                     ]
       ]
```

```
let:[ | Trivial Loops
      [BINDING PROBE HEADs TAILs]
[ | BINDING =
  \[I R].  rec:[ LookUp
                 \[I [F ! A] V].
                    if:[ nil?: F
                                V
                         isLtrl?:F
                            if:[ same?:[I F]  A  V ]
                         isLST?:F
                            LookUp:[ I
                                     [head:F ! head:A]
                                     LookUp:[ I
                                              [tail:F ! tail:A]
                                              V
                                    ]        ]
                         "otherwise"
                            asERROR:["arg" I]
                       ]
                  LookUp:[I R asERROR:["ubi:" I] ]
                 ]
  | PROBE =
  \[N L].
     rec:[ Index
           \[N [V ! Vs]].
               if:[ zero?:N
                          V
                    isLST?:Vs
                      Index:[dcr:N Vs]
                      asERROR:["prb" ! Vs]
                  ]

           if:[ all?:[ pos?:N  isLST?:L]
                  Index:[N L]
                  asERROR:["prb" ! Vs]
              ]
          ]
```

```
  | HEADs =
  fix:[ Loop !
        \Lst.
         let:[ [[H ! T] ! Lst' ]
               Lst
               if:[ nil?:Lst
                     []
                     [H ! Loop:Lst']
                  ]
       ]      ]
  | TAILs =
  fix:[ Loop !
        \Lst .
         let:[ [[H ! T] ! Lst' ]
               Lst
               if:[ nil?:Lst
                      []
                      [T ! Loop:Lst' ]
                  ]
       ]      ]
]
```

```
rec:[ | Serious loops
      [ VALUE APPLY asOPN LIST LISTFUNCTIONAL ]
[ |VALUE =
\[E R].

 let:[ [E0 ! E1]
       asLST:E

  if:[ isERROR?:E
          asERROR:["val" ! E]
       isDCT?:E
          E
       isNML?:E
          E
       isLST?:E
          E
       isFTN?:E
          asCLOSURE:[ R  E ]
       isIDE?:E
          if:[ VOID?:E1
                  BINDING:[E R]
                  E1
              ]
       isAPL?:E
         (( APPLY:R
          ): VALUE:[E0 R]
          ): VALUE:[E1 R]

     ]]
```

```
  |APPLY =
\R. \U. \V .

  if:[ isERROR?:U
          asERROR:["ftn" ! U]
       isDCT?:U
         (asOPN:U):[R V]

       isNML?:U
         PROBE:[U V]

       isIDE?:U
         if:[ nil?:U
                 []
                 asERROR:["apl" ":" U]
             ]

       isAPL?:U
         asERROR:["apl"]

       isLST?:U
         LISTFUNCTIONAL:[ R  U  V ]

       isFTN?:U
         if:[ isCLOSURE?:U

                 let:[ [ R' X ! E ]
                        exCLOSURE:U
                        VALUE:[ E EXTEND:[X V R'] ]
                       ]

                 let:[ [X ! E]
                        asLST:U
                        VALUE:[ E [X ! V] ]
                       ]
             ]
    ]
```

```
  | asOPN =
\D.
 if:[ LSTdct?:D
        \[R Es]. (LIST:"D"):[Es R ]

      SETdct?:D
        \[R Es]. (LIST:"I"):[Es R]

      CODES?:[D "val"]
        \[R E]. VALUE:[E R]

      CODES?:[D "let"]
        \[R [X E E']] .
           VALUE:[ E'
                   EXTEND:[ X VALUE:[E R] R ]
                  ]

      CODES?:[D "rec"]
        \[R [X E E']] .
           rec:[ R'
                 EXTEND:[ X
                          VALUE:[E R']
                          R
                         ]
                 VALUE:[E' R']
                ]

      CODES?:[D "fix"]
        \[R [X ! E]] .
            fix:[ V
                ! VALUE:[ E
                          EXTEND:[ X V R ]
                         ]
                 ]

      CODES?:[D "inc"]
        \[R V]. CHKN:[ V inc ]

      CODES?:[D "dcr"]
        \[R V]. CHKN:[ V dcr ]

      CODES?:[D "add"]
        \[R V]. CHKNxN?:[ V add ]

     "otherwise"
        \[R V]. D:V
    ]
```

```
| LIST =
\B. fix:[ Loop
        ! \[L R].
            let:[ [E ! L']  L
            let:[ V   VALUE:[E R]
            let:[ Vs  Loop:[L' R]

             if:[ isLST?:L
                    if:[ same?:[L L']
                           [V *]
                         same?:[B "D"]
                           [V ! Vs]
                         same?:[B "I"]
                           {V ! Vs}
                         ]
                  "otherwise"
                    VALUE:[L R]
                 ]]]]
         ]

| LISTFUNCTIONAL =
\[ R F V ].
   let:[ [ Fh ! Ft ] F
   let:[ APPLY' APPLY:[[]![]]

    if:[ nil?:isLST?:V
            asERROR:["f-c"]

         all?:[ same?:[F Ft] nil?:head:V ]
            []

         [ (APPLY':Fh):HEADs:V
         ! (APPLY':Ft):TAILs:V
          ]
        ] ]]
]
VALUE:Z
]]]]
```

Chapter 6

Tutorials on Daisy Programming

This section is a collection of tutorials on Daisy programming techniques. The each tutorial focuses on one method or construct, the examples more broadly illustrate aspects of programming. For instance, in the Multiset Examples subsection there are several nontrivial interactive programs.

The tutorials [when finished] include discussions of multisets and their use; programming on recursive data; implementing systems; and modeling hardware systems; parsing and program manipulation.

6.1 Multiset Examples

For programs dealing with multisets, an idealized view of evaluation is not adequate. Evaluation must be considered operationally as computational effort applied to to produce values. One must also keep in mind that Daisy, like Lisp, manipulates references to objects representing values, that these, hence also their computations, may be shared through the environment bindings. The Daisy Interpretation section discusses language execution at this level.

A multiset is a concurrency construct in which a specified list is ordered as its elements' computations finish. The more effort required to produce a value, the later that value appears in the resultant ordering. Only an approximation of effort can be de-

duced from the text of a program. There is implicit overhead in the interpretation process, maintanence of environments, and multitasking. In addition, there are global parameters governing the granularity and degree of concurrency. Even if an exact measure of cost were to be defined, there is a lack of precision in the process of discovering convergent elements.

The programs that follow were executed under minimal settings for granularity, by invoking

```
Daisy -n 2 -s 2 -m 200000
```

The command argument `-m` calls for 200,000 cells of data. The arguments `-n` and `-s` set tight bounds on multitasking. With both set to `2`, there is fine multitasking granularity. The effects are multiset orderings are most strongly related to actual effort and that multitasking overhead is at its maximum.

Only a small fraction of existing Daisy programs use multisets, and indeterminate applicative programming is largely unexplored territory. Although experimentation is encouraged, it would be tedious to reproduce these examples by hand. Their source is available with the installation of this documentation. The base operations used are briefly discussed as they arise. Detailed explanations are given in Daisy Operations section.

6.1.1 A First Example

The following program was suggested by John O'Donnell as a first multiset example. The functions `up` and `dn` compute the distance between two numbers. The expression `up:[N M]` counts how many times `N` must be incremented to reach `M`; `dn:[N M]` counts

how many times N must be decremented to reach M.

```
                                |
up = \[K L].                    | K, L are numerals
  if:[ zero?:K                  | 'zero?' tests for 0
         L                      |
         up:[ inc:K  inc:L ]    | 'inc' increments
      ]                         |
                                |
dn = \[K L].                    | K, L are numerals
  if:[ zero?:K                  |
         L                      |
         dn:[ dcr:K  inc:L ]    | 'dcr' decrements
      ]                         |
                                |
```

The function ABS computes the absolute value of N by concurrently calculating its distances from 0.

```
                                |
ABS = \N.                       | N a numeral
   head:set:[ dn:[N 0]          |
              up:[N 0]          |
             ]                  |
                                |
```

ABS:93 returns 93; ABS:0 returns 0; ABS:-2001 returns 2001. The head operation retrieves the first element of the concurrently ordered list. Assuming a perfect implementation of arithmetic, one of dn:[N 0] or up:[N 0] diverges unless N is 0, in which case both return 0). Hence, the right choice appears at the multiset's head.

The actual implementation of arithmetic is not perfect; numbers are finitely represented and arithmetic operations "wrap around." The expression dn:[-6 0] *eventually* converges—in fact, it returns -6 under the two's complement interpretation used in Daisy. Hence, there can be no guarantee—only an extreme liklihood—that ABS returns the right answer. However, dn:[N 0] returns more quickly when N is positive, and up:[N 0] returns more quickly when N is negative. That is, computing the distance in the right direction requires less computational effort, although the difference diminishes as N approaches the largest representable number. (See Technical Note 1–1).

6.1.2 Implementing Effort

For later examples, a synthetic implementation of effort is needed.

```
                              |
run = ^\[N M]. N:[M*]         | N a numeral, M a value
                              |
```

The function run returns M after N units of effort. It does so by retrieving the Nth element of a cyclic list of Ms. Storage access is a dominant factor gauging work done, and run is a fairly accurate instrument. It executes iteratively, consuming a small, constant amount of storage, independent of N. The actual effort is proportional to N because that many list access are executed by the numeric probe. We also need to implement infinite effort, or divergence. The function dvg runs indefinately by looking for a 0 in a cyclic list of 1s. Like run, dvg is a tight loop.

```
                              |
dvg = ^\[]. in?:[0 [1*]]      | 'in?' tests for
                              |       membership
```

The following program is a multiplication function, which returns 0 should either of its operands be 0.

```
Multiply = \[N M].
  head:set:[ if:[ zero?:N 0  dvg:[] ]
             if:[ zero?:M 0  dvg:[] ]
             mpy:[N M]
            ]
```

Should both of Multiply's operands be non-zero, the two conditional expressions diverge, leaving mpy:[N M] as the only expression whose value can be promoted. Should N be zero, either the first or third expression can produce a value at the head of the multiset. It doesn't matter which, because both expressions return 0. In cases that M and N diverge, Multiply may still produce a result: Multiply:[0 dvg:[]] and Multiply:[dvg:[] 0] both yield 0.

6.1.3 A Concurrent Conditional

In the article "A Note on Conditional Expressions," (*Comm. ACM* **21**, 11 (November, 1978), 931-933) Friedman and Wise discuss the

elimination of conditional tests and their distribution through structure. At times, one would like to reason that an expression like

$$\textit{if } P \textit{ then } A \textit{ else } A$$

is equivalent to A, and further, that

$$\textit{if } P \textit{ then} < E_0 \;!\; E_1 > \textit{else} < E_0' \;!\; E_1' >$$

is equivalent to

```
[ if P then E0 else E0'
! if P then E0 else E0'
]
```

These transformations are invalid for the usual conditional, which is strict in its test. They are inconsistent with the law holding that `if:`$<\Omega\ E\ E'>$ is equivalent to Ω, regardless of E and E'.

The conditional developed here supports the weaker interpretation. The function `GUARD` returns a value after verifying a sequence of tests. Should any of the tests fail, `GUARD` diverges.

```
                                 |
GUARD = ^\[T ! Ts].              | T is a test.
                                 |
  let:[ [V ! Ts'] Ts             | V is either the answer,
                                 |      or another test.
                                 |
   if:[ nil?:T    dvg:[]         | 'nil?' tests for Nil.
        nil?:Ts'  V              |
                  GUARD:Ts       |
      ]]                         |
                                 |
```

There must be at least one test initially, and Nil represents a test that fails. A `let` form is used to identify as `V` the element following `T` and as `Ts'` the ensuing suffix. If `V` is last element of a list, it is taken as the resulting value; otherwise, it is the next test.

The concurrent conditional `CF` looks at three cases (the version published by Friedman and Wise is more elaborite). It tries the standard conditional. At the same time, it checks whether the alternatives, `U` and `V`, are identical. If so, either `U` or `V` can be returned. If both `U` and `V` are lists, the selection can be distributed

to their components.

```
                                |
CF = ^\[T U V].                 | T a test,
                                | U, V alternative values
  let:[ [Uh ! Ut] U             |
  let:[ [Vh ! Vt] V             |
                                |
  head:                         |
   set:[ GUARD:[ "true"         | "true" always succeeds
                 if:[T U V]     |
                ]               |
         GUARD:[ same?:[U V]    | 'same?' tests equality
                 U              |         of reference
                ]               |
         GUARD:[ isLST?:U       | 'isLST?' tests for a list.
                 isLST?:V
                 [ CF:[T Uh Vh] ! CF:[T Ut Vt] ]
                ]
        }]]                     |
                                |
```

The nested let-expressions, identifying the components of U and V might have been written as one:

```
let:[ [[Uh ! Ut] [Vh ! Vt]]
      [    U         V    ]
      head:set:[cdots]
     ]
```

This identification might have been placed within the scope of the third GUARD where it is known that U and V are lists. However, most Daisy programmers prefer to assemble local bindings outside the body of a function. In cases where either U or V is atomic, the bindings are erroneous, but the program is non-erroneous because the names are used. The namings are not essential; writing head:U for Uh, and so on, would not alter the CF's meaning.

The expression

```
CF:[ "true"
       [1 2 [3 4 5] 6]
       [1 2 [3 4 5] 7]
     ]
```

returns `[1 2 [3 4 5] 6]`. If `CF` had been `if`, one could be sure that the result and the second argument were identical; that is, the same list object. With `CF`, the result might be a copy. The expression below also returns `[1 2 [3 4 5] 6]`.

```
CF:[ run:[10000 "true"]
       [1 2 [3 4 5] 6]
       [1 2 [3 4 5] 7]
     ]
```

In this second case, it is fairly certain that a copy is returned because a display of the answer pauses before the '6':

`[1 2 [3 4 5`—*pause*—`] 6]`

which indicates that the outcome of the test is not used until this point, and hence that the result is a copy of the original list [See Technical Note 3–1]. The expression

```
CF:[ dvg:[]
       [1 2 [3 4 5] 6]
       [1 2 [3 4 5] 7]
     ]
```

yields the value `[1 2 [3 4 5]` ⊥`]`, the divergent test not coming into play until the alternatives differ. Finally,

```
CF:[ dvg:[]
       [1 2 [3 4 5] 6]
       [1 2 [3 4 5] 6]
     ]
```

returns `[1 2 [3 4 5] 6]`; `CF` does not need the outcome of the test. Though `CF` supports the distributive law

$$\begin{array}{l} [\ \ \textit{if } P \textit{ then } E_0 \textit{ else } E_0' \\ !\ \ \textit{if } P \textit{ then } E_0 \textit{ else } E_0' \\ ] \end{array}$$

it is not operationally equivalent. Should P be an expression, it might be evaluated twice if distributed according to the law. In `CF`, only a *reference* to the test is distributed; the branches of computation share any progress in its evaluation through `T`'s binding.

6.1.4 A Timely Merge

The word *stream* was first coined by Landin ("A Correspondence between ALGOL 60 and Church's Lambda Notation," *Comm. ACM* 21, 11, 931–933), but now has a variety of meanings. As used here, it refers to a possibly nonterminating sequence of values, accessed in order, and represented by a list. The program below collates two lists, building a single list of their elements. Considered as a function on streams, `MERGE` joins inputs `As` and `Bs` in a single output.

```
                                        |
MERGE = ^\[ As Bs ].                    |
  let:[ [ Ah ! At] As                   | As is a stream
  let:[ [ Bh ! Bt] Bs                   | Bs is a stream
                                        |
  head:                                 |
   set:[crc_hd:[Ah ! MERGE:[Bs At]]     | 'crc_hd' see
         crc_hd:[Bh ! MERGE:[As Bt]]    |          below
        }                               |
                                        |
       ]]                               |
                                        |
```

As earlier, the multiset implements a choice among alternatives. Here, they are to inject the initial element, either of `As` or `Bs`, into the output. In the first case, the head of stream `As` is injected and merging resumes on `Bs` and `As`' tail. However, that result can arise *only after* `Ah` is present. The operation `crc_hd` expects a list and returns that list *once* its head is present. It is essentially `(\L. if:[ head:L  L  L ])`.

`MERGE` expects two nonterminating streams, and produces a nonterminating stream. In the experiments that follow, input streams are developed like this:

```
[run*]:[ ^[ 15  12 215  10  10  15  13 10000*]
         ^[ a1  a2  a3  a4  a5  a6  a7    a8*]
```

The result of this expression is `[a1 a2 ··· a8 a8 ··· ]` **. According**

to the meaning of list application, an equivalent, expression is

```
[ run:[    15 "a1"]
  run:[    12 "a2"]
  run:[   215 "a3"]
      ⋮
  run:[10000 "a8"]
  run:[10000 "a8"]
      ⋮
]
```

That is, an effort is associated with each value by mapping `run` **through a list of numbers. For instance, it takes** `15` **units of effort to produce** `a1` **and** `215` **units of effort to produce** `a3`**.**

`MERGE` **deals with nonterminating streams, The function** `PREFIX` **below returns a the first** `N` **elements of a list** `Vs`**. [See Technical Note 4–1.].**

```
                                          |
PREFIX = ^\N.\Vs.                         |
  rec:[ LOOP                              |
        \[N [V ! Vs]].                    | 'zero?' tests for 0
           if:[ zero?:N                   | 'dcr' decrements.
                  []
                  [V ! LOOP:[dcr:N Vs]]
               ]
        LOOP:[N Vs]                       |
       ]                                  |
                                          |
```

The local function `LOOP` **builds the desired sublist. For example,** `(PREFIX:2):^[A B C D]` **returns** `[A B]`**.**

The expression below merges streams of `as` **and** `bs`**. Of interest is how well the result reflects the effort required to produce an individual element.**

```
(PREFIX:20):
  MERGE:[ [run*]:[ ^[ 15   12 215   10   10   15   13 10000*]
                   ^[ a1   a2  a3   a4   a5   a6   a7    a8*]
                 ]
          [run*]:[ ^[210   11   11 300   11   11   11 10000*]
                   ^[ b1   b2  b3   b4   b5   b6   b7    b8*]
                 ]
        ]
```

The result (it might be different on successive executions) is

```
[ a1 a2 b1 b2 b3 a3 a4 a5 a6 a7
  b4 b5 b6 b7 a8 b8 a8 b8 a8 b8
 ]
```

This shows that `MERGE` is *timely,* in the sense that its output reflects the relative effort producing the inputs. The `crc_hd` operation, or something like it, is essential in obtaining this kind of behavior. So too is the role of the environment in developing shared references. The expression `[Bh ! MERGE:[As Bt]]` converges with constant effort, returning a list of two computations. A version of `MERGE` without `crc_hd` is equally likely to choose either alternative, independent of the work needed to compute the heads. On the same inputs, a `crc_hd`-less version produces the prefix

```
[ a1 b1 a2 b2 a3 b3 a4 b4 a5 b5
  b6 b7 a6 b8 a7 a8 b8 b8 a8 b8
 ]
```

The expression `crc_hd:[Bh ! MERGE:[As Bt]]` has the same value unless `Bh` diverges; and the effort needed to evaluate this expression is a little more than that needed to evaluate `Bh`.

The notion of *fairness* for multisets is the degree of independence between the ordering expression and the ordering of the result. That is, multisets are (more) fair when equal efforts have (more nearly) equal liklihood of advancing in the ordering. What we have called "timeliness" is the (possibly equivalent) notion that values advance (nearly) in proportion to their effort. The `MERGE` examples suggest that this is so, although multiset evaluation is not perfect in this respect. Even if it were, this property may not be inherited by programs using them. The order of the `let` bindings makes `Ah` harder to look up than `Bh`. Since `Ah` is harder to look up, the alternative

```
crc_hd:[Ah ! MERGE:[Bs At]]
```

implicitly requires more effort. In the recursive call to `MERGE`, the two streams are alternated. The object is to eliminate the bias just discussed. It is difficult to tell if the solution works. Experimentation convinces the author that it helps, although, the bias only shows up in degenerate cases. Where there is any appreciable effort in computing the streams, `MERGE` produces a reasonable output stream whether or not altenation is used.

6.1.5 Governing Concurrency

The function `RUN1`, defined in a moment, is a specialized version of `MERGE`. It's purpose is to process a stream of computations so long as a given resource holds out. The resource is represented by a background computation. The assignments below name constants to be used in `RUN1`.

```
                                          |
&DONE = "/DONE/"                          | Reserved symbol
&STOP = "/STOP/"                          | Reserved symbol
STOP? = ^\V. same?:[V &STOP]              | Test for &STOP
run&stop = ^\N. run:[N &STOP]             | Makes a resource
```

The constants `&DONE` and `&STOP` cannot be used in the stream-of-computations called `Work` below. If `RUN1` is viewed as a scheduler, these are priviledged communications. `RUN1` injects the elements of `Work` until the resource `Rsrc` is exhausted. Like `MERGE`, it produces a stream of results. Unlike `MERGE`, it detects stream termination, although this does not mean that `Work` *must* be finite.

```
                                          |
RUN1 = ^\[Rsrc Work].                     |
                                          |
  let:[ [W ! Ws]  Work                    | Work is a stream
  let:[ [X Y]     {run:[1 W]  Rsrc}       |
                                          |
   if:[ nil?:Work  [&DONE]                |
        STOP?:X    [X]                    |
                   [X ! RUN1:[Y Ws]]      |
       ]]]                                |
                                          |
```

The outer `let`-binding names components of the list `Work`. The inner `let`-binding names the two elements of a multiset value. `X` could bind either to the resource or to `run:[1 W]`, `Y` to the other. Where `Work` terminates, `RUN1` terminates, appending the constant `&DONE` to its output. The second branch of the conditional tests whether `Rsrc` is exhausted. If so, `RUN1` terminates, appending the constant `&STOP`. The test is on `X`, the head of the multiset in which `Rsrc` is competing for the ordering. Where `STOP?:X` fails (and assuming `W` cannot be `&STOP`), `X` must be the first item in `Work` and, therefore, `Y` must be the resource. `RUN1` appends `X` to its output and resumes, running `Ws` against `Y` (i.e. `Rsrc`).

In the latter case, `Y` and `Rsrc` bind to the *same object.* This means that progress in the resource computation is inherited by recursive calls to `RUN1`. Eventually, this computation completes, hence `RUN1` produces a finite list. The expression

```
RUN1:[ run&stop:500
       [run*]:[ ^[100 25 200 15 25 300  5  5  5 400]
                ^[ j1 j2  j3 j4 j5  j6 j7 j8 j9 j10]
              ]
      ]
```

uses the same technique shown with `MERGE` to create a stream of non-trivial computations. The result is a list

```
[j1 j2 j3 j4 /STOP/]
```

With `run&stop:500` replaced by `run&stop:1000`, the result is

```
[j1 j2 j3 j4 j5 j6 j7 j8 /STOP/]
```

With the resource `run&stop:1500`, the result is

```
[j1 j2 j3 j4 j5 j6 j7 j8 j9 j10 /DONE/]
```

In `RUN1`, the multiset `{ run:[1 W] Rsrc }` assures a minimal effort in the input stream. In pathological experiments, it was too hard to tune `Rsrc` computations, and in some trials it seemed that `Rsrc` did not get enough attention to make progress. In reasonable examples, `{W  Rsrc}` was adequate.

About the same methods are used in the variation below, which allows two jobs in `Work` to run concurrently. One difference is that concurrent computations are explicitly tagged, in order to identify them in scheduling. In addition, a messy termination

test is omitted, so that RUN2 expects a nonterminating stream.

```
                                    |
  RUN2 = ^\[Rsrc Work].             |
    let:[ [W0 W1 ! Ws]              |
          Work                      |
                                    |
    let:[ [[X I] ! Etc ]            |
          { crc_hd:[W0   "A"]       |
            crc_hd:[W1   "B"]       |
            crc_hd:[Rsrc "R"]       |
           }                        |
                                    |
     if:[ same?:[I "R"]             |
            [X]                     |
          same?:[I "A"]
            [X ! RUN2:[Rsrc [W1 ! Ws]] ]
          same?:[I "B"]
            [X ! RUN2:[Rsrc [W0 ! Ws]] ]
        ]
                                    |
        ]]                          |
                                    |
```

RUN2 executes the first two elements of Work against the resource. The distinct literals attached to W0, W1, and Rsrc classify the outcome. For instance, 'B' means that the second job in Work concluded, and RUN2 then runs the first and third jobs against Rsrc. As was done in MERGE earlier, crc_hd is used to force a good ordering in the multiset. Since this approach eliminates the need for the symbol &STOP, it might be preferable to that of RUN1. It is also easier to generalize. The expression

```
 RUN2:[ run&stop:500
        [run*]:[ ^[100 25 200 15 25 300  5  5  5 400 10000*]
                 ^[ j1 j2  j3 j4 j5  j6 j7 j8 j9 j10     Z*]
               ]
       ]
```

gives a non-terminating stream to RUN2. The result,

```
[j2 j1 j4 j5 j3 j7 j8 j9 /STOP/]
```

shows that later, shorter jobs advance past earlier, longer jobs.

Should unlimited concurrency be desired, one would again use a test for the reserved value &STOP and simply copy the Work stream into a multiset.

```
                                        |
RUNall = ^\[Rsrc Work].                 |
                                        |
  rec:[ Icopy                           |
        \W. if:[ nil?:W                 |
                   []
                   { head:W ! Icopy:tail:W }
                ]
  rec:[ Watch                           |
        \[W ! Ws]. if:[ STOP?:W         |
                          [W]
                          [W ! Watch:Ws]
                       ]
      | in                              |
        Watch:{ Rsrc ! Icopy:Work }     |
       ]]
```

Icopy takes a list W and builds { 0:W 1:W 2:W ··· }. Though Work may be ordered initially, Icopy:Work is not. Watch watches for termination of the resource, ending the output stream where this happens. RUNall's body adds the resource to the other computations. The expression

```
RUNall:[ run&stop:1500
         [run*]:[ ^[100 25 200 15 25 300  5  5  5 400 10000*]
                  ^[ j1 j2  j3 j4 j5  j6 j7 j8 j9 j10     Z*]
       ]       ]
```

returns [j2 j1 j4 j5 j7 j3 j8 j9 j6 j10 /STOP/].

On the same argument, the three versions of RUN produce the following orderings. RUN1 builds

```
[j1 j2 j3 j4 j5 j6 j7 j8 j9 j10]
```

It preserves the order of Work because individual jobs are run. RUN2 executes two jobs at a time, allowing shorter jobs to advance around longer ones:

```
[j2 j1 j4 j5 j3 j7 j8 j9 j6 j10]
```

RUNall has the same effect with a higher degree of concurrency.

```
[j2 j1 j4 j5 j7 j3 j8 j9 j6 j10]
```

The multiset below also runs the ten jobs concurrently.

```
{ run:[100 "j1"]
  run:[ 25 "j2"]
  run:[200 "j3"]
  run:[ 15 "j4"]
  run:[ 25 "j5"]
  run:[300 "j6"]
  run:[  5 "j7"]
  run:[  5 "j8"]
  run:[  5 "j9"]
  run:[400 "j10"]
}
```

Its result is the list

```
[j7 j4 j5 j2 j8 j9 j1 j3 j6 j10]
```

This is a better ordering, in the sense of approaching the ideal reflection of effort. That j2, j4, and j5 preceed j8 and j9 reflects the overhead in multiset evaluation, which contributes to a bias in favor of earlier expressions. For RUNall the overhead is more significant because Icopy must iterate n times to place $\mathtt{j}_n$ in the "run set."

6.1.6 Concurrent Interaction

The following program monitors an input channel from the operator's key board, returning &STOP where it finds an interrupt character.

```
SENSEOPR = ^\[Prompt InterruptChar].
   rec:[ Loop \[C ! Cs]. if:[ same?:[C InterruptChar]
                              &STOP
                              Loop:Cs
                        ]
         Loop:console:Prompt
       ]
```

The locally defined Loop searches a list for an occurance of InterruptChar, and returns the &STOP constant. The body of SENSEOPR applies Loop

to the stream of input characters created by console; the console operation directs interactive input (stdin in UNIX) into such a stream.

Now, the object is to allow the operator to represent the resource used in the program RUN2, developed in the previous section. The resource expression run&stop:1500 is replaced by a call to SENSEOPR. The same jobs are run, but their efforts are increased ten-fold for this experiment. The operator's prompt is ?? and the interrupt character is #

```
RUN2: [ SENSEOPR:["??" "#"]
        [run*]: [ ^[ 1000 250 2000 150  250
                     3000  50   50  50 4000 10000
                   *]
                  ^[   j1  j2   j3  j4   j5
                       j6  j7   j8  j9  j10     Z
                   *]
          ]         ]
```

The interaction below is taken verbatim from an execution script, from the point that this expression is entered. System utterences are shaded.

```
??
??xxx
 [??xxxxxxx
??xxxxxxxxxx
j2 j1 j4 ??xx
??xx
??xxxxxxx
j5 j3 j7 ??xxxxxxx#xxxxxxxx
j8 j9 /STOP/]
```

The result of the expression is

```
[j2 j1 j4 j5 j3 j7 j8 j9 /STOP/]
```

as it was for the earlier invokation of RUN2. The operator's interaction with the system interleaved with this output. The instance of console raises the prompt '??' after each new-line. Over the course of the program, the operator types as sequence of x's, newlines, and one '#', which stops RUN2. Here are three obeservations about this experiment.

1. **The effort of interpreting SENSEOPR replaces that of interpreting run&stop, providing a means to stop the potentially infinite output.**

2. **The effect of interaction is coursely grained: once a prompt is raised, control is passed to the I/O subsystem and not relinquished until the next *line* of input has been typed. This is not a property just of the console operation, but is also due to host input facilities, which buffer interactive input. The buffering is done both for efficiency and to support "line editing" (e.g. back-spaces).**

3. **Hence, SENSEOPR's effort, as sensed by Daisy, is directly proportional to the number of characters it looks at; and it is independent of the time taken to enter key strokes. The operator creates more effort by typing more characters.**

For UNIX implementations, a version of console exists that records effort while waiting for host input. This operation is liveconsole. To use it, one must turn off host's line-editing features (e.g. by setting the terminal attribute cbreak). Once this is done, and with console reassigned to be the liveconsole operation, the execution of RUN2 can be interrupted by a single key stroke. This is shown in the following recorded session, in which the same experiment is done.

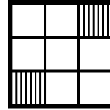
```
& console = liveconsole
 console
& RUN2:[ SENSEOPR:["??" "#" etc. ]
??  [j2 j1 j4 j5 j3 j7 j8 j9  # j6 /STOP/]
```

Here, the operator types a single '#' to interrupt. The prompt is still issued, but eight jobs run before the response. Job j6 completes before the interruption takes effect.

6.1.7 Coordinated Concurrency

Earlier examples use multisets to run concurrent independent tasks. The series of examples below show one way to program dependent computations. Each of the examples involves a collection of tasks seeking paths through a maze. Each position of a maze is repesented by a list of four pointers to its neighbors, with

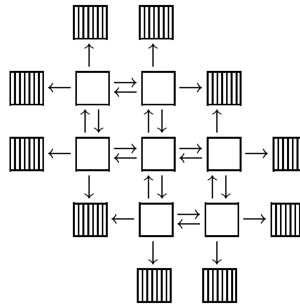
Nil representing a blocked position. An maze is a list of *positions.* **For example, the 3×3 maze**



is expressed as

```
fix:[ [ M00 M01 M02          | [ ][ ][x]
        M10 M11 M12          | [ ][ ][ ]
        M20 M21 M22          | [x][ ][ ]
      ]                      |
    ! [ [  [] M01 M10  [] ]  | = M00
        [  [] M02 M11 M00 ]  | = M01
        []                   | = M02
        [ M00 M11 M20  [] ]  | = M10
        [ M01 M12 M21 M10 ]  | = M11
        [ M02  [] M22 M11 ]  | = M21
        []                   | = M20
        [ M11 M22  [] M20 ]  | = M21
        [ M12  []  [] M21 ]  | = M22
      ]                      |
    ]                        |
```

This expression builds the network



Each of the positions in the maze is simply a list of neighboring positions, with boundaries represented by Nil.

The program `SEEK`**, below, implements a tasks looking for a path in a maze puzzel. Each instance of** `SEEK` **is located at a positon** `Here` **and is attempting to find the destination** `To`**.** `SEEK`

records its path in the accumulator `Path`; in addition it has two parameters, `Been` and `GBeen` telling it what positions have been visited by other tasks.

```
SEEK = ^\[Here To Path Been]. \GBeen.
  let:[ [N E S W]  Here
  let:[ Advice     ADVISE:[Here Been GBeen]

   if:[ nil?:Here
           ["Quit"]
        in?:[Here Been]
           ["Stop"]
        same?:[Here To]
           ["Home" Path]
        "otherwise"
           ["GoOn" Here
                   SEEK:[N To ["N" ! Path] Advice]
                   SEEK:[E To ["E" ! Path] Advice]
                   SEEK:[S To ["S" ! Path] Advice]
                   SEEK:[W To ["W" ! Path] Advice]
            ]
      ] ]]
```

`SEEK` returns one of four *reports* to the supervisory program `MAZE`, below. Should an instance find itself at a blocked position, it returns `[Quit]`. Should its current position have been seen before, it returns `[Stop]`. Should it be at its destination `SEEK` returns the report `[Home` *path*`]`, where *path* is a list of positions. In other cases, `SEEK` returns the form `[GoOn`*position* p_0 p_1 p_2 p_3`]`, giving its current position and four continuations. Each of the continuations has the same destination but starts at an adjacent position—`N`, `E`, `S`, and `W` are like directions on a compass.

`SEEK` passes `Advice` to its successors, which is the result of some functon, `ADVISE`, on `Here`, `Been`, and `GBeen`. Various versions of `ADVISE` are defined later, but each returns a list of visited positions.

The supervisory function `MAZE` is defined below, with explanation notes following. `MAZE` takes a list of visited positions and a

stream of SEEK tasks; it returns a list of paths.

```
MAZE = let:[ [Kill Home Quit Stop GoOn ]   | NOTES
             [  0    1    2    2    3  ]   | [1]
                                            |
    \[Known  Reports].                      | [2]
                                            |
    let:[ [[Cmd A0 A1 A2 A3 A4] ! Others ]  |
          Reports                           |
    let:[ Seen                              |
          [A0 ! Known]                      |
                                            |
    (val:Cmd):                              | [3]
      [ [Cmd]                               | [4] Kill
        [A0 &RTN ! MAZE:[Known Others]]     | [5] Home
        MAZE:[Known Others]                 | [6] Quit, Stop
        MAZE:[ Seen                         | [7] GoOn
               { A1:Seen                    |
                 A2:Seen                    |
                 A3:Seen                    |
                 A4:Seen                    |
               ! Others                     |
               }                            |
       ]       ]                            |
      ]]   ]                                |
```

[1] **MAZE is defined in the scope of a let-expression, binding the commands Kill, Home, Quit, Stop, and GoOn to numbers. This is an encoding technique; see note [3] below.** [2] **The Known argument is a list of positions that have been seen by some instance of SEEK. Reports is a stream of tasks. The first let-binding identifes an element of this stream as a list, consisting of a command, Cmd, followed by as many as five operands.** [3] **Assuming Cmd is one of Kill, Home, Quit, Stop, or GoOn, the expression (val:Cmd) returns a number between 0 and 3—see Note [1] above. The encoding is used to index a list of four alternative continuations for MAZE. The same function results should explicit tests be made. For example, the next line would be**

```
if:[ same?:[Cmd "Kill"]
           [Cmd]
         ⋮
```

[4] In this case, the command is Kill, and MAZE terminates in the sense that it returns the finite list [Kill], ending the stream. SEEK does not generate a command of this form. We shall see in a moment that this command results when an external resource is exhausted. [5] In this case, the instruction is [Home *path*], that is, some instance of SEEK has found a path to its destination. MAZE issues that path (&RTN is a new-line) and resumes supervising the remaining tasks. [6] If the command is [Quit] or [Stop], MAZE resumes supervising the remaining tasks, with nothing to report in its output stream. [7] In this case, the command is GoOn, its first operand, A0, is a position, and A1 through A4 are functions,

\GBeen. ⋯

each expecting a list of positions. The innermost let-expression identifies as Seen the list of Known positions with A0 added. MAZE updates its global list of positions and also passes it to each of the tasks it adds to the running set.

Let us summarize the system described. The supervisory program MAZE manages a stream of concurrent tasks, each reporting information about the ongoing maze search. These reports, or "system calls," can result in five actions.

1. A Kill command terminates the system.
2. A Quit or Stop command terminates a tasks.
3. A Home command adds a discovered path to MAZE's output stream.
4. A GoOn command adds new SEEK tasks to the running set.
5. A GoOn command adds to the list of positions known to have been visited.

An instance of SEEK receives information about the global search state from two sources. In SEEK, the expressions

SEEK:[*position* To *path* Advice]

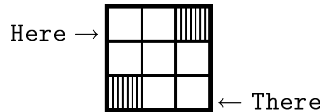
provide, through parameter Been, what is known of the maze by the immediate anscestor. In MAZE, the expressions A_i:Seen provide, through parameter GBeen, collective knowledge of search. It

is the function ADVISE, yet to be defined, that determines what knowledge is used by an individual. The program TEST runs the MAZE system on a given $N \times N$ maze.

```
TEST = ^\[N NxNPuzzel Time].

  let:[ Here
        0:NxNPuzzel
  let:[ There
       (dcr:mpy:[N N]):NxNPuzzel

        MAZE:[ []
               { (SEEK:[ Here There [] [] ]):[]
                 dn:[Time ["Kill" ]]
                }
              ]

      ]]
```

With N given, TEST locates the upper-left and lower-right positions of NxNPuzzel, calling them Here and There. It invokes MAZE with an initially empty list of known positions and a multiset of two tasks. One of these is a SEEK, seeking a path from Here to There; the other is a timer-task, set to issue a Kill instruction after a given Time.

Below are discussed three executions on TheMaze, a 3×3 puzzle



In the first, all advice is ignored. This is accomplished by defining

```
ADVISE = ^\[Here Been Gbeen]. []
```

The expression TEST:[3 TheMaze 1000] yields

```
[[S E S N S E]
 [S E W E S N S E]
 [S E S N W E S N S E]
 [S E S N S N S N W E S N S E]
 [S E N S S N S N S N S N W E S N S E]
 [S E S N S N S N S N S N S N S N W E S N S E]
 ⋮
```

Read the paths from right to left. This test shows the preference for going north and east, as might be inferred from the definitions of SEEK **and** MAZE**. Specifically,** MAZE **builds its schedule as**

```
{ A1:Seen  A2:Seen  A3:Seen  A4:Seen  !  Others}
```

where Others **is bound to the set of pending tasks.** *Evaluation of the literal* Others *adds overhead to the pending tasks; and this overhead accumulates.* **In fact, the trial shown above failed to terminate, suggesting that the overhead subsumes any progress on the timer-tasks. Hence, it may be wrong, or at least unfair, to add tasks at the head of "run set." A solution to this problem is developed later.**

A second definition of ADVISE **incorporates collective global knowledge about the search.**

```
ADVISE = ^\[Here Been Gbeen]. Gbeen
```

In each instance of SEEK**, the** Advice **used is that provided by** MAZE**. The expression** TEST:[3 TheMAZE 3000] **returns**

```
[[S E S E]
 [S E E S]
 [E S E S]
 [E S S E]
 Kill]
```

A third test on the identical puzzle uses a version of ADVISE **in which the information used is that provided by creating instance of** SEEK**.**

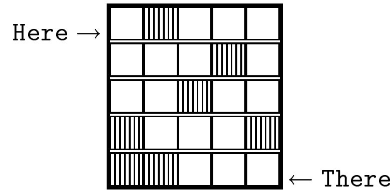
```
ADVISE = ^\[Here Been Gbeen]. [Here ! Been]
```

This time, TEST:[3 TheMAZE 3000] **returns**

```
[[S E S E]
 [E S E S]
 [E S S E]
 [S E E S]
 Kill]
```

In neither case is the searching fully coordinated; in particular, each path found passes through the center square. The following

programs show that there is, in fact, a difference between the two forms of ADVICE. This time, a 5×5 puzzle is used:



With global advice, GBeen, the expression TEST:[5 TheMAZE 100000] **returns**

```
[ [E S E E S E S S]
  [E S E E S S E S]
  [E E S E S E S S]
  [E E S E S S E S]
 Kill
]
```

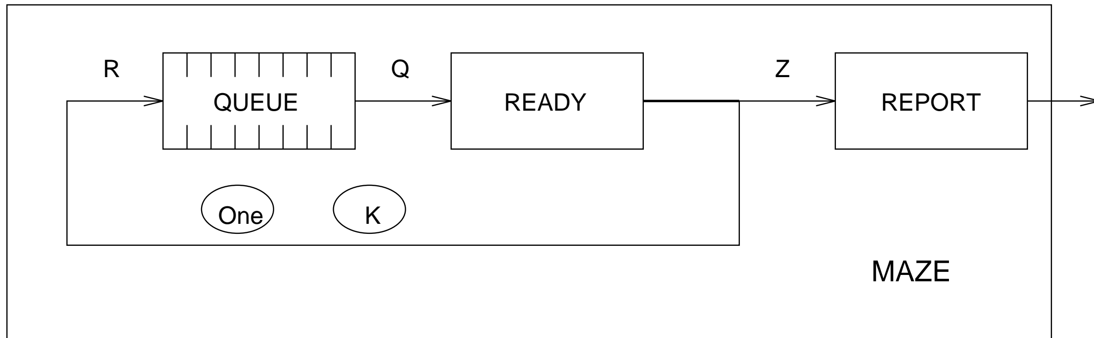
With local advice, [Here ! Been], **the same expression returns**

```
[ [E S E E S E S S]
  [E E S E S E S S]
  [E E S E S S E S]
  [E S E E S S E S]
  [E S S W S S E E N E N E S S]
  [E E S W S W S S E E N E N E S S]
  [E S S W S S E E N E E S]
  [E E S W S W S S E E N E E S]
 Kill
]
```

Let us now return to the question of fairness in scheduling these tasks. It was noted that adding tasks to the head of the multiset resulted in a last-in-first-out bias. In one case, there was an indication of starvation—of the timer, in fact—where tasks are added too frequently.

To make the multiset of SEEK tasks more queue-like, we must arrange to add them at the tail of the multiset. Working top-down, the MAZE program becomes a system of simultaneous list

definitions, according to the diagram



Each of the main functions, `QUEUE`, `READY`, and `REPORT`, iterates over nonterminating linear lists, named `Q`, `R`, and `Z`. `READY` produces a list-of-pairs, which must be split into a pair of lists; that is the meaning of the transposition symbol `(X)` in the diagram. The initial tasks on the list `R` are a timer and a `SEEK` instance, depicted as `--[K]---[One]--`. In Daisy, the system is

```
MAZE = ^\[Time One].                          |
  let:[ K                                     |
        dn:[Time ["Kill"]]                    |
                                              |
  rec:[ [Q [Z R]]                             |
        [                                     |
          QUEUE:R                             | = Q
          xps:READY:[[] {One K ! Q} ]         | = [Z R]
         ]                                    |
        REPORT:Z                              |
      ]]                                      |
```

The important subprogram is the task manipulator `READY`, which is much like the earlier version of `MAZE`. `READY` issues pairs of values, each consisting of an external communication and a new task.

READY's

```
READY = let:[ [Kill Home Skip Quit Stop GoOn ]
              [   0    2    1    1    1    3 ]

    \[Known  Reports].

      let:[ [[Cmd A0 A1 A2 A3 A4] ! Others ] Reports
      let:[ Seen [A0 ! Known]

 (val:Cmd):
     [ [ [Cmd  ["Skip"]] ! []]                      | Kill
                                                    |
       READY:[Known Others]                         | Quit,Stop,Idle
                                                    |
       [ [ A0  ["Skip"]] ! READY:[Known Others] ]   | Home
                                                    |
       [ ["."  A1:Seen ]                            | GoOn
         ["."  A2:Seen ]                            |
         ["."  A3:Seen ]                            |
         ["."  A4:Seen ] ! READY:[Seen Others] ]    |
                                                    |
      ] ]]   ]
```

When READY has no external message it pairs a '.' with the next task. A trivial [Skip] task is used when there is a message. As before, the [Kill] command terminates the task stream. In MAZE, the expression

```
xps:READY:[[] {One K ! Q} ]   | = [Z R]
```

transposes READY's output into the lists Z and R. The transposition function used is

```
xps = [(\x.x) *]
```

The task list R feeds back into READY having passed through QUEUE, which simply turns R into a multiset.

```
QUEUE = ^\[J ! Js]. {J ! QUEUE:Js}
```

The REPORT program issues the list Z of reports, omitting the '.'

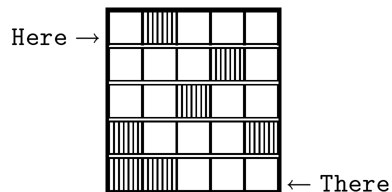
place holders.

```
REPORT = ^\[V ! Vs].
  let:[ NewLine
        NmlAsChr:10
  let:[ R
        if:[nil?:Vs  []  REPORT:Vs]

   if:[ same?:[V "."]
          R
          [V NewLine ! R]
       ]]
```

The TEST **program is like before. It takes an $N \times N$ puzzle, locates the starting and finishing positions, and invokes** MAZE**. The version below includes a provision for interruption, using the** RUN1 **and** SENSEOPR **functions defined earlier.**

```
TEST = ^\[N NxNPuzzel Time].
  let:[ Here
        0:NxNPuzzel
  let:[ There
        (dcr:mpy:[N N]):NxNPuzzel

  (\W. RUN1:[SENSEOPR:["??" "] W]):

  MAZE:[ Time
         (SEEK:[ Here There [] [] ]):[]
        ]
      ]]
```

With "global" advice and again using the puzzel



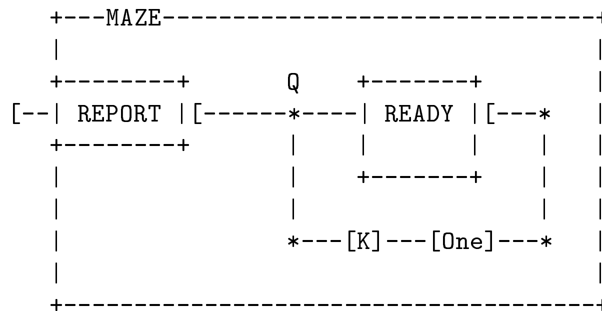
The expression test produces the paths

```
[ [E S E E S S E S]
  [E E S E S S E S]
  [E S E E S E S S]
  [E E S E S E S S]
  Kill
 ]
```

Coding READY **as a two output function centralizes the command interpretation, but the "plumbing" is simplified if the interpretation is distributed. The** MAZE **system**

```
MAZE = ^\[Time One].
  let:[ K
        run:[Time ["Kill"]]

  rec:[ Q
        {One K ! READY:[[] Q ]}
        REPORT:Q
      ]]
```

has the diagram



```
   +---MAZE------------------------------+
   |                                     |
   +---------+       Q    +-------+      |
[--| REPORT |[------*----| READY |[---*  |
   +---------+       |    |       |   |  |
   |                 |    +-------+   |  |
   |                 |                |  |
   |                 *---[K]---[One]---*  |
   |                                     |
   +-------------------------------------+
```

In this version, READY **simply ignores any command that does not generate new tasks; and it subsumes the role of** QUEUE **by building**

a multiset.

```
READY = let:[ [Kill Home Skip Quit Stop GoOn ]
              [   0    2    1    1    1    3 ]
  \[Known  Reports].
   let:[ [[Cmd A0 A1 A2 A3 A4] ! Others ] Reports
   let:[ Seen [A0 ! Known]

(val:Cmd):
    [ [["Skip"]]                | Kill
      READY:[Known Others]      | Quit,Stop,Idle
      READY:[Known Others]      | Home
      { A1:Seen                 | GoOn
        A2:Seen                 |
        A3:Seen                 |
        A4:Seen                 |
      ! READY:[Seen Others]     |
      }                         |
     ] ]]   ]                   |
```

The `REPORT` program is also a command interpreter, which passes `[Home` *path*`]` and `[Kill]`, ignoring everything else. The translation of `[Kill]` to `&STOP` is done for the sake of `RUN1`, as before.

```
REPORT = let:[ [Kill Home Skip Quit Stop GoOn ]
               [   0    2    1    1    1    3 ]
  \Reports.
   let:[ [[Cmd A0 A1 A2 A3 A4] ! Others ] Reports

(val:Cmd):
    [ [&STOP]                           | Kill
      REPORT:Others                     | Quit,Stop,Idle
       [ A0 &NEWLINE ! REPORT:Others ]  | Home
      REPORT:Others                     | GoOn
     ] ]]
```

6.2 Technical Notes

NOTE **1–1 In the current implementation, this version of `ABS` never fails to give the right answer, although it might fail to terminate. Tail recursive programs do not operate in bounded space, and an expression like `up:[1 0]` exhausts storage: there are not as**

many list cells as there are numbers from `1` **up to** `0`**. However, a truely iterative version of** `up` **could be written, as is shown in the next section.** NOTE **3–1 The expected output would be**

[1 2 [3 4 5] —*pause*—6]

The pause occurs before the '`]`**' because of strictness in the** `issue` **operation. This matter is discussed in the Daisy Operations section, in the description of** `issue`**. To explore the next two examples, the author wrote another, lazier, version of issue:**

```
myissue = ^\[V Etc].
  if:[ isAtm?:V
       [ V ! Etc]
       [ "["       ! myissue:[
         head:V    [
         "!"       ! myissue:[
         tail:V    [
         "]"       !
         Etc       ]]]]]
        ]
}
```

This program gives an in-order expansion of a list structure. Unlike `issue`**, it does not de-reference interior cells.** NOTE **4–1** `PREFIX`

takes its arguments one at a time merely to avoid dangling list delimiters. One can write

(PREFIX:20): *Interesting expression*

instead of

```
PREFIX:[ 20
         Interesting expression
       ]
```