Tables

Icon's `table` data type can be thought of as an array that can be subscripted with values of any type.

The built-in function `table` is used to create a table:

```lisp
][ t := table();
    r := T1:[] (table)
```

To store values in a table, simply assign to an element specified by a subscript (sometimes called a `key`):

```lisp
][ t[1000] := "x";
    r := "x" (string)

][ t[3.0] := "three";
    r := "three" (string)

][ t["abc"] := [1];
    r := L1:[1] (list)
```

Values are referenced by subscripting.

```lisp
][ t["abc"];
    r := L1:[1] (list)

][ t[1000];
    r := "x" (string)
```
Tables, continued

Tables can't be output with write(), but Image can describe the contents of a table:

```plaintext
write(Image(t));
T1: [
  1000->"x",
  3.0->"three",
  "abc"->L1:[1]
]
```

Assigning a value using an existing key simply causes the old value to be replaced:

```plaintext
T1: [
  t[3.0] := "Here's 3.0";
  r := "Here's 3.0"  (string)
]

T1: [
  t["abc"] := "xyz";
  r := "xyz"  (string)
]

T1: [
  t[1000] := &null;
  r := &null  (null)
]

write(Image(t));
T2: [
  1000->&null,
  3.0->"Here's 3.0",
  "abc"->"xyz"
]
```
Tables, continued

If a non-existent key is specified, the table's default value is produced. The default default-value is \&null:

```
][ t := table();  
    r := T1:[]   (table)
]

][ t[999];  
    r := \&null   (null)
```

A default value may be specified as the argument to table:

```
][ t2 := table(0);  
    r := T1:[]   (table)
]

][ t2["xyz"];  
    r := 0       (integer)

][ t2["abc"] += 1;  
    r := 1       (integer)

][ t2["abc"];  
    r := 1       (integer)

][ t3 := table("not found");  
    r := T1:[]   (table)
]

][ t3[50];  
    r := "not found"  (string)
```

Language design issue: References to non-existent list elements fail, but references to non-existent table elements succeed and produce an object that can be assigned to. Is that good or bad?
Tables, continued

A key quantity represented with multiple types produces multiple key/value pairs.

```
][ t := table();
   r := T1:[] (table)

][ t[1] := "integer";
   r := "integer" (string)

][ t["1"] := "string";
   r := "string" (string)

][ t[1.0] := "real";
   r := "real" (string)

][ write(Image(t));
   T1:[
      1->"integer",
      1.0->"real",
      "1"->"string"]

][ t[1];
   r := "integer" (string)

][ t["1"];
   r := "string" (string)
```

Be wary of using reals as table keys. Example:

```
][ t[1.0000000000000001];
   r := &null (null)

][ t[1.00000000000000001];
   r := "real" (string)
```
Table application: word usage counter

A simple program to count the number of occurrences of each "word" read from standard input:

```
link split, image
procedure main()
    wordcounts := table(0)

    while line := read() do
        every word := !split(line) do
            wordcounts[word] +:= 1

        write(Image(wordcounts))
    end
```

Interaction:

```
% wordtab
to be or
not to be
^D
T1: [
    "be"->2,
    "not"->1,
    "or"->1,
    "to"->2]
```

Question: How could we also print the number of distinct words found in the input?

Image is great for debugging, but not suitable for end-user output.
Table sorting

Applying the `sort` function to a table produces a list consisting of two-element lists holding key/value pairs.

Example:

```lisp
][  write(Image(wordcounts));
T1:[
  "be"->2,
  "not"->1,
  "or"->1,
  "to"->2]
]
][  write(Image(sort(wordcounts)));
L1:[
  L2:["be", 2],
  L3:["not", 1],
  L4:["or", 1],
  L5:["to", 2]]
```

`sort` takes an integer-valued second argument that defaults to 1, indicating to produce a list sorted by keys. An argument of 2 produces a list sorted by values:

```lisp
][  write(Image(sort(wordcounts,2)));
L1:[
  L2:["not", 1],
  L3:["or", 1],
  L4:["to", 2],
  L5:["be", 2]]
```

`sort`'s second argument may also be 3 or 4, which produces "flattened" versions of the results produced with 1 or 2, respectively.
Table sorting, continued

An improved version of wordtab that uses sort:

```plaintext
link split, image
procedure main()
    wordcounts := table(0)
    while line := read() do
        every word := !split(line) do
            wordcounts[word] += 1

        pairs := sort(wordcounts, 2)
        every pair := !pairs do
            write(pair[1], "\t", pair[2])
    end

Output:

    not   1
    or    1
    to    2
    be    2

Problem: Print the most frequent words first rather than last.
```
Tables—default value pitfall

Recall this pitfall with the list(N, value) function:

```clojure
][ list(5,[]);
   r1 := L1:[L2:[],L2,L2,L2,L2]  (list)
```

There is a similar pitfall with tables:

If [] is specified as the default value, all references to non-existent keys produce the same list.

Example:

```clojure
][ t := table([]);
   r := T1:[]  (table)

][ put(t"x", 1);

][ put(t"y", 2);

][ t"x";
   r := L1:[1,2]  (list)

][ t"y";
   r := L1:[1,2]  (list)

][ [t"x", t"y"];
   r := L1:[L2:[1,2],L2]  (list)

][ [t"x", t"y", t"z"];
   r := L1:[L2:[1,2],L2,L2]  (list)
```

Solution: Stay tuned!
Table application: Cross reference

Consider a program that prints a cross reference listing that shows the lines on which each word appears.

```
% xref
to be or
not to be is not
going to be
the question
^D
be............1 2 3
going...........3
is.............2
not............2 2
or..............1
question.......4
the.............4
to.............1 2 3
```

Problem: Sketch out a solution.
Cross reference solution

```plaintext
procedure main()
    refs := table()
    line_num := 0

    while line := read() do {
        line_num +:= 1
        every w := !split(line) do {
            /refs[w] := []
            put(refs[w], line_num)
        }
    }

    every pair := !sort(refs) do {
        writes(left(pair[1],15,"."))
        every writes(!pair[2]," ")
        write()
    }
end
```

Question: Are lists really needed in this solution?

Another approach:

```plaintext
procedure main()
    refs := table([])  # BE CAREFUL!
    line_num := 0

    while line := read() do {
        line_num +:= 1

        every w := !split(line) do {
            refs[w] ||:= [line_num]
        }
    }

    ...
end
```
Tables and generation

When applied to a table, \(!\) generates the values in the table.

Consider a table \(\text{romans}\) that maps roman numerals to integers:

\[
\begin{array}{l}
\text{write(Image(romans));} \\
\text{T1: [} \\
\text{ "I"->1,} \\
\text{ "V"->5,} \\
\text{ "X"->10]} \\
\text{ every } \text{!romans;} \\
10 \ (\text{integer}) \\
1 \ (\text{integer}) \\
5 \ (\text{integer})
\end{array}
\]

The \(\text{key(t)}\) function generates the keys in table \(t\):

\[
\begin{array}{l}
\text{every } \text{key(romans);} \\
\text{ "X" \ (string)} \\
\text{ "I" \ (string)} \\
\text{ "V" \ (string)}
\end{array}
\]

\[
\begin{array}{l}
\text{every } \text{romans[key(romans)];} \\
10 \ (\text{integer}) \\
1 \ (\text{integer}) \\
5 \ (\text{integer})
\end{array}
\]

Language design question: What is the Right Thing for \(!t\) to generate?
Table key types

Any type can be used as a table key.

```plaintext
[[ t := table();

[[ A := [];  
[[ B := "b"];

[[ t[A] := 10;
[[ t[B] := 20;
[[ t[t] := t;

[[ write(Image(t));
T2:[
   L1:[]->10,
   L2:[
      "b"]->20,
   T2->T2]
```

Table lookup is identical to comparison with the `===` operator, using value semantics for scalar types and reference semantics for structure types.

```plaintext
[[ A;
   r := L3:[] (list)
[[ t[A];
   r := 10 (integer)
[[ t[[]];
   r := &null (null)

[[ get(B);
   r := "b" (string)
[[ B;
   r := L3:[] (list)
[[ t[B];
   r := 20 (integer)
```
Table application: Cyclic list counter

Consider a procedure \( \text{lists}(L) \) to count the number of unique lists in a potentially cyclic list:

\[
\begin{align*}
\text{lists}([]); & \quad r := 1 \quad \text{(integer)} \\
\text{lists}([[],[]]); & \quad r := 3 \quad \text{(integer)} \\
A := []; & \\
\text{put}(A,A); & \\
\text{put}(A,[A]); & \\
A; & \quad r := L1:[L1,L2:[L1]] \quad \text{(list)} \\
\text{lists}(A); & \quad r := 2 \quad \text{(integer)}
\end{align*}
\]

Implementation:

\[
\begin{align*}
\text{procedure lists}(L, \text{seen}) & \\
& / \text{seen} := \text{table}() \\
& \quad \text{if} \ \backslash \text{seen}[L] \ \text{then return 0} \\
& \quad \text{count} := 1 \\
& \quad \text{seen}[L] := 1 \quad \# \ \text{any non-null value would do} \\
& \quad \text{every e := !L & type(e) == "list" do} \\
& \quad \quad \text{count} +:= \text{lists}(e, \text{seen}) \\
& \quad \text{return count}
\end{align*}
\]

Problems: Write \( \text{lcopy}(L) \) and \( \text{lcompare}(L1,L2) \), to copy and compare lists.
csets—sets of characters

Icon's cset data type is used to represent sets of characters.

In strings, the order of the characters is important, but in a cset, only membership is significant.

A cset literal is specified using apostrophes. Characters in a cset are shown in collating order:

```
]c 'abcd';
  r := 'abcd'  (cset)

]c 'bcad';
  r := 'abcd'  (cset)

]c 'babccabc';
  r := 'abc'  (cset)

]c 'babccabdbaab';
  r := 'abcd'  (cset)
```

Equality of csets is based only on membership:

```
]c 'abcd' === 'bcad' === 'bcbbbbabcd';
  r := 'abcd'  (cset)
```

(In other words, csets have value semantics.)

If c is a cset, *c produces the number of characters in the set.

For !c, the cset is converted to a string and then characters are generated.
csets, continued

Strings are freely converted to character sets and vice-versa.

The second argument for the `split` procedure is actually a character set, not a string. Because of the automatic conversion, this works:

```plaintext
split("...1..3..45,78,,9 10   ", ",., ")
```

But more properly it is this:

```plaintext
split("...1..3..45,78,,9 10   ", ',., ')
```

Curio: Converting a string to a cset and back sorts the characters and removes the letters.

```plaintext
][  string(cset("tim korb"));
    r := " bikmort"  (string)
```
csets, continued

A number of keywords provide handy csets:

```
[[ write(&digits);
  0123456789
  r := &digits  (cset)

[[ write(&lcase);
  abcdefghijklmnopqrstuvwxyz
  r := &lcase  (cset)

[[ write(&ucase);
  ABCDEFGHIJKLMNOPQRSTUVWXYZ
  r := &ucase  (cset)

Others:
  &ascii           The 128 ASCII characters
  &cset            All 256 characters in Icon's "world"
  &letters        The union of &lcase and &ucase
```
csets, continued

The operations of union, intersection, difference, and complement (with respect to &cset) are available on csets:

\[
\begin{align*}
&[\ 'abc' \ ++ \ 'cde'; \quad \# \text{union} \\
&\quad r := 'abcde' \ (\text{cset}) \\
\end{align*}
\]

\[
\begin{align*}
&[\ 'abc' \ ** \ 'cde'; \quad \# \text{intersection} \\
&\quad r := 'c' \ (\text{cset}) \\
\end{align*}
\]

\[
\begin{align*}
&[\ 'abc' \ -- \ 'cde'; \quad \# \text{difference} \\
&\quad r := 'ab' \ (\text{cset}) \\
\end{align*}
\]

\[
\begin{align*}
&[\ ^\sim'abc'; \quad \# \text{complement} \\
&\quad r := 253 \ (\text{integer}) \\
\end{align*}
\]

Problem: Create csets representing the characters that may occur in:

(a) A real literal

(b) A Java identifier

(c) A UNIX filename

Problem: Print characters in string \texttt{s1} that are not in string \texttt{s2}.
csets, continued

Problem: Using csets, write a program to read standard input and calculate the number of distinct characters encountered.

Problem: Print the numbers in this string (s).

On February 14, 1912, Arizona became the 48th state.
Sets

A set can be created with the set(L) function, which accepts a list of initial values for the set:

```plaintext
[] [ s := set([1,2,3]);
    r := S1:[2,1,3]  (set)

][ s2 := set(["x", 1, 2, "y", 1, 2, 3, "x"]);
    r := S1:[2,"x",1,3,"y"]  (set)

][ s3 := set(split("to be or not to be"));
    r := S1:['to','or','not','be']  (set)

][ set([],[],[]);
    r := S1:[L1:[]],[L2:[]],[L3:[]]  (set)

][ s4 := set();
    r := S1:[]  (set)
```

Values in a set are unordered. All values are unique, using the same notion of equality as the === operator.

The unary *, !, and ? operators do what you'd expect:

```plaintext
[] [ *s2;
    r := 5  (integer)

][ .every !s;
    2  (integer)
    1  (integer)
    3  (integer)

][ ?s2;
    r := "y"  (string)
```

Sets were a late addition to the language.
Sets, continued

The \texttt{insert}(S, x) function adds the value \( x \) to the set \( S \), if not already present, and returns \( S \). It always succeeds.

The \texttt{delete}(S, x) function removes the value \( x \) from \( S \) and returns \( S \). It always succeeds.

The \texttt{member}(S, x) function succeeds iff \( S \) contains \( x \).

Examples:

\begin{verbatim}
][ every insert(s, !"testing");
  Failure

][ s;
  r := S1: ["s", "e", "g", "t", "i", "n"] (set)

][ insert(s, "s");
  r := S1: ["s", "e", "g", "t", "i", "n"] (set)

][ every delete(s, !"aieou");
  Failure

][ s;
  r := S1: ["s", "g", "t", "n"] (set)

][ member(s, "a");
  Failure

][ member(s, "t");
  r := "t" (string)
\end{verbatim}
Sets, continued

Set union, intersection, and difference are supported:

```icon
[[ fives := set([5,10,15,20,25]);
  r := S1:[5,10,15,20,25]  (set)
]

[[ tens := set([10,20,30]);
  r := S1:[10,20,30]  (set)
]

[[ fives ** tens;
  r := S1:[10,20]  (set)
]

[[ fives ++ tens;
  r := S1:[5,10,15,20,25,30]  (set)
]

[[ fives -- tens;
  r := S1:[5,15,25]  (set)
]

[[ tens -- fives;
  r := S1:[30]  (set)
]
```

Problem: Write a program that reads an Icon program on standard input and prints the unique identifiers. Assume that `reserved()` generates a list of reserved words such as "if" and "while", which should not be printed.
Sets and tables—common functions

The insert, delete, and member functions can be applied to tables:

```
[ t := table();
  r := T1:[] (table)
]
[ t["x"] := 10;
  r := 10 (integer)
]
[ insert(t, "v", 5);
  r := T1:{"v"->5,"x"->10} (table)
]
[ member(t, "i");
  Failure
]
[ delete(t, "v");
  r := T1:{"x"->10} (table)
]
```

Note that the only way to truly delete a value from a table is with the delete function:

```
[ t["x"] := &null;  # the key remains...
  r := &null (null)
]
[ t;
  r := T1:{"x"->&null} (table)
]
[ delete(t, "x");
  r := T1:[] (table)
]```
Records

Icon provides a record data type that is simply an aggregate of named fields.

A record declaration names the record and the fields. Examples:

\[
\text{record name(first, middle, last)}
\]

\[
\text{record point(x, y)}
\]

\textit{record declarations are global and appear at file scope.}

A record is created by calling the record constructor.

\[
\text{p := point(3, 4);}
\]
\[
r := \text{R1:point}_1(3, 4) \quad \text{(point)}
\]

\[
\text{type(p);}
\]
\[
r := "point" \quad \text{(string)}
\]

\[
\text{p.x;}
\]
\[
r := 3 \quad \text{(integer)}
\]

\[
\text{p.y;}
\]
\[
r := 4 \quad \text{(integer)}
\]

\[
\text{p2 := point(, 3);}
\]
\[
r := \text{R1:point}_3(\text{&null}, 3) \quad \text{(point)}
\]

\[
\text{type(point);}
\]
\[
r1 := "procedure" \quad \text{(string)}
\]

\[
\text{image(point);}
\]
\[
r2 := "record constructor point" \quad \text{(string)}
\]
Records, continued

A simple example:

```plaintext
record point(x, y)
record line(a, b)

procedure main()
    A := point(0,0)
    B := point(3,4)

    AB := line(A,B)
    write("Length: ", length(AB))

    move(A,-3,-4)
    write("New length: ", length(AB))
end

procedure length(ln)
    return sqrt((ln.a.x-ln.b.x)^2 +
               (ln.a.y-ln.b.y)^2)
end

procedure move(p, dx, dy)
    p.x +:= dx
    p.y +:= dy
end
```

Output:

```
Length: 5.0
New length: 10.0
```

Problem: Modify `move()` so that a new point is created, rather than modifying the referenced point.
Records, continued

A routine to produce a string representation of a point:

```plaintext
procedure ptos(p)
    return "(" || p.x || "," || p.y || ")"
end
```

Records can be meaningfully sorted with `sortf`:

```plaintext
pts := [point(0,1), point(2,0), point(-3,4)];

every write(ptos(!sortf(pts,1)));
(-3,4)
(0,1)
(2,0)
Failure

every write(ptos(!sortf(pts,2)));
(2,0)
(0,1)
(-3,4)
Failure
```

Fields in a record can be accessed with a subscript:

```plaintext
pt := point(3,4);

pt[2];
r := 4 (integer)
```