# Lists

Icon has a `list` data type. Lists hold sequences of values.

One way to create a list:

```
][ [1,2,3];
   r := [1,2,3]  (list)
```

A given list may hold values of differing types:

```
][ L := [1, "two", 3.0, []];
   r := [1,"two",3.0,[]]  (list)
```

An element of a list may be referenced by subscripting:

```
][ L[1];
   r := 1  (integer)

][ L[2];
   r := "two"  (string)

][ L[-1];
   r := []  (list)

][ L[10];
Failure
```

The other way to create a list:

```
][ list(5, "a");
   r := ["a","a","a","a","a"]  (list)
```

# Lists, continued

A *list section* may be obtained by specifying two positions:

```
][ L := [1, "two", 3.0, []];
   r := [1,"two",3.0,[]]  (list)

][ L[1:3];
   r := [1,"two"]  (list)

][ L[2:0];
   r := ["two",3.0,[]]  (list)
```

Note the asymmetry between subscripting and sectioning: subscripting produces an element, sectioning produces a list.

```
][ L[2:3];
   r := ["two"]  (list)

][ L[2];
   r := "two"  (string)

][ L[2:2];
   r := []  (list)
```

Contrast with strings:

```
][ s := "123";
   r := "123"  (string)

][ s[2:3];
   r := "2"  (string)

][ s[2:2];
   r := ""  (string)
```

Question: What is the necessary source of this asymmetry?

# Lists, continued

Recall `L`:

```
][ L;
   r := [1,"two",3.0,[]]  (list)
```

Lists may be concatenated with `|||`:

```
][ [1] ||| [2] ||| [3];
   r := [1,2,3]  (list)

][ L[1:3] ||| L[2:0];
   r := [1,"two","two",3.0,[]]  (list)
```

Concatenating lists is like concatenating strings—a new list is formed:

```
][ L := [1, "two", 3.0, []];
   r := [1,"two",3.0,[]]  (list)

][ L := L ||| [9999] ||| L ||| [];
   r := [1,"two",3.0,[],9999,1,"two",3.0,[]]
```

For the code below, what is the final value of `nines`?

```
nines := []
every nines |||:= |[9] \ 7
```

# Lists, continued

The number of top-level elements in a list may be calculated with `*`:

```
][ L := [1, "two", 3.0, []];
   r := [1,"two",3.0,[]]  (list)

][ *L;
   r := 4  (integer)

][ *[];
   r := 0  (integer)
```

Problem: What is the value of the following expressions?

```
*[[1,2,3]]

*[L, L,[[]]]

*[,,]

**[[],[]]

*(list(1000000,0) ||| list(1000000,1))
```

# Lists, continued

List elements can be changed via assignment:

```
][ L := [1,2,3];
   r := [1,2,3]  (list)

][ L[1] := 10;
   r := 10  (integer)

][ L[-1] := "last element";
   r := "last element"  (string)

][ L;
   r := [10,2,"last element"]  (list)
```

List sections <u>cannot</u> be assigned to:

```
][ L[1:3] := [];
Run-time error 111
variable expected
...
```

Problem: Write a procedure `assign(L1, i, j, L2)` that approximates the operation `L1[i:j] := L2`.

# Complex subscripts and sections

Lists within lists can be referenced by a series of subscripts:

```
][ L := [1,[10,20],[30,40]];
   r := [1,[10,20],[30,40]]  (list)

][ L[2];
   r := [10,20]  (list)

][ L[2][1];
   r := 10  (integer)

][ L[2][1] := "abc";
   r := "abc"  (string)

][ L;
   r := [1,["abc",20],[30,40] ]  (list)
```

A series of subscripting operations to reference a substring of a string-valued second-level list element:

```
][ L[2][1];
   r := "abc"  (string)

][ L[2][1][2:0] := "pes";
   r := "pes"  (string)

][ L;
   r := [1,["apes",20],[30,40]]  (list)

][ every write(!L[2][1][2:4]);
p
e
Failure
```

# Lists as stacks and queues

The functions `push`, `pop`, `put`, `get`, and `pull` provide access to lists as if they were stacks, queues, and double-ended queues.

`push(L, expr)` adds `expr` to the left end of list `L` and returns `L` as its result:

```
][ L := [];
   r := []  (list)

][ push(L, 1);
   r := [1]  (list)

][ L;
   r := [1]  (list)

][ push(L, 2);
   r := [2,1]  (list)

][ push(L, 3);
   r := [3,2,1]  (list)

][ L;
   r := [3,2,1]  (list)
```

# Lists as stacks and queues, continued

`pop(L)` removes the leftmost element of the list `L` and returns that value. `pop(L)` fails if `L` is empty.

```
][ L;
   r := [3,2,1]  (list)

][ while e := pop(L) do
...      write(e);
3
2
1
Failure

][ L;
   r := []  (list)
```

Note that the series of pops clears the list.

A program to print the lines in a file in reverse order:

```
procedure main()
    L  := []
    while push(L, read())
    while write(pop(L))

end
```

With generators:

```
procedure main()
    L := []
    every push(L, !&input)
    every write(!L)
end
```

# Lists as stacks and queues, continued

`push` returns its first argument:

```
][ x := push(push(push([],10),20),30);
   r := [30,20,10]  (list)

][ x;
   r := [30,20,10]  (list)
```

`put(L, expr)` adds `expr` to the right end of `L` and returns `L` as its result:

```
][ L := ["a"];
   r := ["a"]  (list)

][ put(L, "b");
   r := ["a","b"]  (list)

][ every put(L, 1 to 3);
Failure

][ L;
   r := ["a","b",1,2,3]  (list)
```

# Lists as stacks and queues, continued

`get(L)`, performs the same operation as `pop(L)`, removing the leftmost element of the list `L` and returning that value. `get(L)` fails if `L` is empty.

Yet another way to print the numbers from 1 to 10:

```
L := []
every put(L, 1 to 10)
while write(get(L))
```

`pull(L)` removes the rightmost element of the list `L` and returns that value. `pull(L)` fails if `L` is empty.

```
][ L := [1,2,3,4];
   r := [1,2,3,4]  (list)

][ while write(pull(L));
4
3
2
1
Failure

][ L;
   r := []  (list)
```

Any of the five functions `push`, `pop`, `put`, `get`, and `pull` can be used in any combination on any list.

A visual summary:

```
push  ==>            ==> pull
pop   <==  List     <== put
get   <==
```

# List element generation

When applied to lists, `!` generates elements:

```
][ .every ![1, 2, ["a", "b"], 3.0, write];
   1  (integer)
   2  (integer)
   ["a","b"]  (list)
   3.0  (real)
   function write  (procedure)
```

Problem: Write a procedure `common(L1, L2, L3)` that succeeds if the three lists have an integer value in common. Easy: Assume that the lists contain only integers. Hard: Don't assume that.

Problem: Write procedures `explode(s)` and `implode(L)` such as those found in ML.

```
][ explode("test");
   r := ["t","e","s","t"]  (list)

][ implode(r);
   r := "test"  (string)
```

# Sorting lists

The function `sort(L)` produces a sorted copy of the list `L`. `L` is not changed.

```
][ L := [5,1,10,7,-15];
   r := [5,1,10,7,-15]  (list)

][ Ls := sort(L);
   r := [-15,1,5,7,10]  (list)

][ L;
   r := [5,1,10,7,-15]  (list)

][ Ls;
   r := [-15,1,5,7,10]  (list)
```

Lists need not be homogeneous to be sorted:

```
][ sort(["a", 10, "b", 1, 2.0, &null]);
   r := [&null,1,10,2.0,"a","b"]  (list)
```

Values are ordered first by type, then by value. Page 161 in the text shows the type ordering used for heterogenous lists.

A program to sort lines of standard input:

```
procedure main()
    L := []
    while put(L, read())
    every write(!sort(L))
end
```

Problem: Describe two distinct ways to sort lines in descending order.

# Sorting lists, continued

Sorting a list of lists orders the lists according to their order of creation—not usually very useful.

The `sortf(L, i)` function sorts a list of lists according to the `i`-th element of each list:

```
][ L := [[1, "one"], [8, "eight"], [2, "two"]];
   r := [[1,"one"], [8,"eight"], [2,"two"]]

][ sortf(L, 1);
   r := [[1,"one"],[2,"two"],[8,"eight"]]

][ sortf(L, 2);
   r := [[8,"eight"], [1,"one"], [2,"two"]]
```

The value `i` can be negative, but not zero.

Lists without an `i`-th element sort ahead of other lists.

# Lists in a nutshell

- Create with `[expr, ...]` and `list(N, value)`
- Index and section like strings
  Can't assign to sections
- Size and element generation like strings
- Concatenate with `|||`
- Stack/queue access with `put`, `push`, `get`, `pop`, `pull`
  Parameters are consistent: list first, then value
- Sort with `sort` and `sortf`

Challenge:

Find another language where equivalent functionality can be described as briefly.

# Reference semantics for lists

Some types in Icon use *value semantics* and others use *reference semantics*.

Strings use value semantics:

```
][ s1 := "string 1";
   r := "string 1"  (string)

][ s2 := s1;
   r := "string 1"  (string)

][ s2[1] := "x";
   r := "x"  (string)

][ s1;
   r := "string 1"  (string)
][ s2;
   r := "xtring 1"  (string)
```

Lists use reference semantics:

```
][ L1 := [1,2,3];
   r := [1,2,3]  (list)

][ L2 := L1;
   r := [1,2,3]  (list)

][ L2[1] := "x";
   r := "x"  (string)

][ L1;
   r := ["x",2,3]  (list)

][ L2;
   r := ["x",2,3]  (list)
```

# Reference semantics for lists, continued

Earlier examples of list operations with `ie` have been edited. What `ie` really shows for list values:

```
][ lst1 := [1,2,3];
   r := L1:[1,2,3]  (list)

][ lst2 := [[],[],[]];
   r := L1:[L2:[],L3:[],L4:[]]  (list)
```

The `Ln` tags are used to help identify lists that appear multiple times:

```
][ [lst1, lst1, [lst1]];
   r := L1:[L2:[1,2,3],L2,L3:[L2]]  (list)
```

Consider this:

```
][ lst := [1,2];
   r := L1:[1,2]  (list)

][ lst[1] := lst;
   r := L1:[L1,2]  (list)
```

Then this:

```
][ lst[1][2] := 10;
   r := 10  (integer)

][ lst;
   r := L1:[L1,10]  (list)
```

# Reference semantics for lists, continued

More:

```
][ X := [1,2,3];
   r := L1:[1,2,3]  (list)

][ push(X,X);
   r := L1:[L1,1,2,3]  (list)

][ put(X,X);
   r := L1:[L1,1,2,3,L1]  (list)

][ X[3] := [[X]];
   r := L1:[L2:[L3:[L3,1,L1,3,L3]]]  (list)

][ X;
   r := L1:[L1,1,L2:[L3:[L1]],3,L1]  (list)
```

Explain this:

```
][ L := list(5,[]);
   r := L1:[L2:[],L2,L2,L2,L2]  (list)
```

# Reference semantics for lists, continued

An important aspect of list semantics is that equality of two list-valued expressions is based on whether the expressions reference the same list object in memory.

```
][ lst1 := [1,2,3];
   r := L1:[1,2,3]  (list)

][ lst2 := lst1;
   r := L1:[1,2,3]  (list)

][ lst1 === lst2;
   r := L1:[1,2,3]  (list)

][ lst2 === [1,2,3];
Failure

][ [1,2,3] === [1,2,3];
Failure

][ [] === [];
Failure
```

# Reference semantics for lists, continued

Icon uses call-by-value for transmission of argument values to a procedure.

However, an argument is a type such as a list, which uses reference semantics, the value passed is a reference to the list itself. Changes made to the list will be visible to the caller.

An extension of the procedure `double` to handle lists:

```
procedure double(x)
    if type(x) == "string" then
        return x || x
    else if numeric(x) then
        return 2 * x
    else if type(x) == "list" then {
        every i := 1 to *x do
            x[i] := double(x[i])
        return x
        }
    else
        fail
end
```

Usage: (note that `L` is changed)

```
][ L := [3, "abc", [4.5, ["xx"]]];
   r := [3, "abc", [4.5, ["xx"]]]  (list)

][ double(L);
   r := [6, "abcabc", [9.0, ["xxxx"]]]  (list)

][ L;
   r := [6, "abcabc", [9.0, ["xxxx"]]]  (list)
```

# `image` and `Image`

Lists cannot be output with the `write` function. To output lists, the `image` and `Image` routines may be used.

The built-in function `image(X)` produces a string representation of any value:

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

][ image("s");
   r := "\"s\""  (string)

][ write(image("s"));
"s"
   r := "\"s\""  (string)

][ image(write);
   r := "function write"  (string)

][ image([1,2,3]);
   r := "list_13(3)"  (string)
```

For lists, `image` only shows a "serial number" and the size.

# `image` and `Image`, continued

The Icon procedure `Image` can be used to produce a complete description of a list (or any value):

```
][ write(Image([1,2,[],4]));
L3:[
  1,
  2,
  L4:[],
  4]
   r := "L3:[\n  1,\n  2,\n  L4:[],\n  4]"
```

Note that `Image` produces a string, which in this case contains characters for formatting.

An optional second argument of 3 causes `Image` to produce a string with no formatting characters:

```
][ write(Image([1,2,[],4], 3));
L8:[1,2,L9:[],4]
   r := "L8:[1,2,L9:[],4]"  (string)
```

`Image` is not a built-in function; it must be linked:

```
link image
procedure main()
   ...
end
```

# Simple text processing with `split`

A number of text processing problems can be addressed with a simple concept: splitting a line into pieces based on delimiters and then processing those pieces.

There is a procedure `split(s, delims)` that returns a list consisting of the portions of the string `s` delimited by characters in `delims`:

```
][ split("just a test here ", " ");
   r := ["just","a","test","here"]  (list)

][ split("...1..3..45,78,,9 10  ", "., ");
   r := ["1","3","45","78","9","10"]  (list)
```

`split` is not a built-in function; it must be linked:

```
link split
procedure main()
  ...
end
```

# `split`, continued

Consider a file whose lines consist of zero or more integers separated by white space:

```
5 10 0 50

200
1 2 3 4 5 6 7 8 9 10
```

A program to sum the numbers in such a file:

```
link split
procedure main()
    sum := 0
    while line := read() do {
        nums := split(line, " \t")
        every num := !nums do
            sum +:= num
        }

    write("The sum is ", sum)
end
```

Problem: Trim down that flabby code!

```
procedure main()
    sum := 0

    write("The sum is ", sum)
end
```

If `split` has a third argument that is non-null, both delimited and delimiting pieces of the string are produced:

```
][ split("520-621-6613", "-", 1);
   r := ["520","-","621","-","6613"]  (list)
```

# `split`, continued

Write a procedure `extract(s, m, n)` that extracts a portion of a string `s` that represents a hierarchical data structure. `m` is a major index and `n` is a minor index. Major sections of the string are delimited by slashes and are composed of minor sections separated by colons. Here is a sample string:

**`/a:b/apple:orange/10:2:4/xyz/`**

It has four major sections which in turn have two, two, three and one minor sections.

A call such as `extract(s, 3, 2)` locates the third major section ("`10:2:4`") and return the second minor section ("`2`").

```
][ extract(s, 1, 2);
   r := "b"  (string)

][ extract(s, 4, 1);
   r := "xyz"  (string)

][ extract(s, 4, 2);
Failure
```

# Command line arguments

The command line arguments for an Icon program are passed to `main` as a list of strings.

```
procedure main(a)
    write(*a, " arguments:")
    every write(image(!a))
end
```

Execution:

```
% args just "a test" right here
4 arguments:
"just"
"a test"
"right"
"here"
% args
0 arguments:
%
```

Problem: Write a program `picklines` that reads lines from standard input and prints ranges of lines specified by command line arguments. Lines may be referenced from the end of file, with the last line being -1.

Examples:

```
picklines 1 2 3 2 1 < somefile

picklines 1..10 30 40 50 < somefile

picklines 1..10 -10..-1 < somefile
```

# picklines—Solution

```
link split
procedure main(args)
    lines := []
    while put(lines, read())

    picks := []
    every spec := !args do {
        w := split(spec, ".")
        every put(picks, lines[w[1]:w[-1]+1])
        }

    every write(!!picks)
end
```