Lists

Lists are a built-in Icon datatype. Lists can be accessed from the beginning (the way you would in LISP, Prolog, etc), the end, or indexed (the way you would access an array in Pascal).

- Lists can be heterogeneous, they can contain elements of different type.

```text
x := ["hello", 1, 3.14, "x", "y"] A list of a string, an integer, a float, and two strings.
y := list(5, "hej") A list of five strings: ["hej", ..., "hej"].
```

Tables

Tables are associative arrays, they map keys to values. Both values and keys can be of arbitrary type.

- `x:=table(0)` Create a new table x whose default value is 0. This means that if you look up a key which has no corresponding value, 0 is returned.
- `*x` Number of elements in the table.
- `?x` An arbitrary element from the table.
- `keys(x)` Generate all keys in x, one at a time.
- `!x` Generate all values, one at a time.

```text
every X := !L do
   write(X)
```
Tables II – Examples

```
x["monkey"] := "banana"
x[3.14] := "pi"
x["pi"] := 3.14
x["pi"] += 1  Increment pi by 1
r := x["coconut"]  r will be 0
member(x, 3.14) returns "pi"
member(x, "banana") fails
insert(x, "banana", 5)  x["banana"] := 5
delete(x, "monkey") remove "monkey"
every m := key(x) do write(m) write keys
every m := !x do write(m) write values
```

Sets

- Sets are unordered collections of elements.

```
x := set([5, 3, "monkey"])  Create a 3-element set from a list.
member(x, 5) returns 5
member(x, "banana") fails
insert(x, "banana") add "banana" to x
delete(x, 5) returns the set {3, "banana", "monkey"}
```

Sets...

- *x* number of elements (3)
- ?x random element from x
- !x generate the elements
- $S := S_1 \ op \ S_2$ set union ($\ op=++$), intersection ($\ op=**$), difference ($\ op=--$).
- while insert($S$, read($f$))  Read elements from file $f$ into set $S$

Records

- Records and procedures are the only declarations in Icon. They must be declared at the outermost (global) level.
- You don’t give the types of the fields, just their names.
- $\text{type}(X)$, where $X$ is a record variable, will return the name (a string) of the record type.

```
record complex (re, im)
procedure P ()
  local x, r, i
  x := complex(5, 4)
  r := x.re  # or r := x[1]
  i := x.im  # or r := x[2]
  t := type(x)  # t="complex"
end
```
Indirect Procedure Calls

Procedure names can be constructed at runtime, allowing a powerful form of indirect procedure call.

Remember to include the directive `invocable all` at the beginning of your module.

`proc(P)` returns the procedure whose name is the string `P`.

```icon
P1 := proc("MyProc1")
P2 := proc("MyProc" || "2")
P3 := proc("find")  // Built-ins OK, too.
P4 := proc("**", 2)  // Multiplication has arity 2.
L := [P1, P2, P3, P4]  // A list of procedures.
L[2](45, "X2")  // Calling MyProc2(45, "X2").
```

Binary Trees in Icon

```icon
link ximage
record node (item,left,right)
procedure Preorder (T)
  if \T then {
    write(T.item);
    Preorder(T.left); Preorder(T.right)
  }
end

procedure main()
  t := node(1, node(2, &null, &null),
            node(3, &null,
                 node(4, &null, &null)))
  Preorder(t); xdump(t)
end
```

```
> icont b
> b
1
2
3
4
R_node_4 := node()
  R_node_4.item := 1
R_node_4.left := R_node_1 := node()
  R_node_1.item := 2
R_node_4.right := R_node_3 := node()
  R_node_3.item := 3
  R_node_3.right := R_node_2 := node()
    R_node_2.item := 4
```