CSc 372

Comparative Programming Languages

31 : Icon — Data Structures

Christian Collberg

collberg+372@gmail.com

Department of Computer Science

University of Arizona

Data Structures

- Icon has built-in support for records, lists, tables, and sets.
- These data structures can be freely combined, so that it is easy to construct a list of tables of sets, ...

Records

Records

- Records and procedures are the only declarations in Icon. They must be declared at the outermost (global) level:

  ```
  record name(field1,field2,...)
  ```

- You don't give the types of the fields, just their names.
- `type(X)`, where `X` is a record variable, will return the name (a string) of the record type.
- If `R` is a record variable, `R.field1` references the field whose name is `field1`.

Complex Arithmetic Module

```
record complex(re, im)

procedure add(a,b)
   return complex(a.re+b.re, a.im+b.im)
end

procedure main ()
   local x, r, i
   x := complex(5, 4)
   y := complex(1,2)
   z := add(x,y)

   r := z.re      # or r := z[1]
   i := z.im      # or r := z[2]
   t := type(z) # t="complex"
end
```

Lists

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.

`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"]`.

`x[2:4]` The list consisting of the second, third, and fourth element of `x`.

Lists vs. Strings

- Lists are indexed in the same way as strings:

$$\begin{array}{ccccccc} [& & "h", & & 42 & &] \\ & \uparrow & & \uparrow & & \uparrow & \\ & 1 & & 2 & & 3 & \end{array}$$

- Strings are immutable. This means that when you assign to an element of a string you actually get a new string as result.
- Lists are mutable. That is, when you assign to an element of a list, the list actually changes.

List Operations

`s := list()` Create an empty list.

`s := list(n)` Create a list of n `null`s.

`s := list(n,v)` Create a list of n `v`s.

`s := *x` Number of elements of `x`.

`x ||| y` Concatenate `x` and `y`.

`!x` Generate all elements of the list, in order, as in `every X := !L do write(X)`.

Examples

```
][ L := list(5,10);
   r1 := L1:[10,10,10,10,10]   (list)
][ L[2] := 42;
][ L;
   r3 := L1:[10,42,10,10,10]
][ L := [1,2,3,4,5];
][ L[1:3];
   r5 := L1:[1,2]
][ L[0:-3];
   r6 := L1:[3,4,5]
][ every i := !L do write(i);
1
2
3
4
5
```

List Operations...

`x ||| y` Concatenate `x` and `y`.

`put(x, 67)` Add `67` to the end of the list `x`.

`get(x)` Remove and return the last element of `x`.

`push(x, 1024)` Add a new element to the beginning of `x`.

`pop(x)` Remove and return the first element of `x`.

`!x` Generate all elements of the list, in order, as in `every X := !L do write(X)`.

`?x` Return a random element from list.

`x===y` Succeed if `x` and `y` are the same string.

`x~===y` Succeed if `x` and `y` are different strings.

Examples

```
][ L := [[1,2],[3,4],[5,6]];
][ L[2,1] := 42;
][ L;
   r3 := L1:[L2:[1,2],L3:[42,4],L4:[5,6]]
][ x := pop(L);
][ x;
   r5 := L1:[1,2]  (list)
][ L;
   r6 := L1:[L2:[42,4],L3:[5,6]]
][ L := [1,2,3,4,5];
][ every !L :=: ?L;
][ L;
   r9 := L1:[2,1,5,3,4]
```

Fibonacci

```
procedure main()
    n := 20
    f := [1,1]
    repeat {
        i := get(f)
        if i>n then break
        write(i)
        put(f,i+f[1])
    }
end
```

Prime Sieve

```
procedure main()
   n := 100
   p := list(n,1)
   every i := 2 to sqrt(n) do
      if p[i]=1 then
         every j := i+i to n by i do
            p[j] := 0
   every i := 2 to n do
      if p[i]=1 then
         write(i)
end
```

Tables

Tables

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

Table Operations

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 correspoding 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.

```
every X := keys(T) do
   write(X, " ==> ", T[X])
```

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

- Sets are unordered collections of elements.
- `set()` creates an empty set.
- `set(L)` creates a set from a list of elements.
- All the standard set-operations (intersection, etc.) are built-in.

Set Operations

`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"`}

`*x` number of elements (3)

`?x` random element from `x`

`!x` generate the elements

Set Operations

`S := S1 op S2` set union (op=`++`), intersection (op=`**`), difference (op=`--`).

`while insert(S, read(f))` Read elements from file `f` into set `S`

Prime Sieve

```
procedure main()
   n := 100
   p := set()
   every i:=2 to n do insert(p,i)
   every i := 2 to sqrt(n) &
              member(p,i) &
              j := i+i to n by i do
                 delete(p,j)
   every i := 2 to n & member(p,i) do
      write(i)
end
```

Binary Trees

Binary Trees in 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
```

Binary Trees in Icon. . .

```
> 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
```

Readings and References

- Read `Christopher, pp 29--34,105--126`.

Acknowledgments

- Some material on these slides has been modified from Thomas W Christopher's Icon Programming Language Handbook,

  `http://www.tools-of-computing.com/tc/CS/iconprog.pdf`.