CSc 372 — Comparative Programming Languages

33 : Icon — Generators

Christian Collberg
Department of Computer Science
University of Arizona
collberg+372@gmail.com

November 30, 2005

1 Expressions as Generators

- Icon expressions are generators, they can return a sequence of values.
- Every expression has three possibilities: It can generate
  1. no values (≡ failure),
  2. one value, or
  3. several values.

2 Expressions as Generators...

- Icon has many built-in generators, e.g. `i to j by k`. The following two statements are equivalent:

```
every i := j to k do p(i)
every p(j to k)
```

- `every e` asks `e` to generate as many values as it possibly can, by backtracking into it until it fails.
- `every e1 do e2` evaluates `e2` for every value generated by `e1`.

3 Expressions as Generators...

- The number of values a generator will produce depends on the environment in which it is invoked:

```
][ write(1 to 5);
1
   r1 := 1  (integer)
][ every write(1 to 5);
1
2
3
4
5
Failure
```

4 find

- find(e1, e2) returns the positions within the string e2 where the string e1 occurs.
- find("wh", "who, what, when") has three possible solutions and hence generates three values.

```
#                      123456789012345
][ every i:=find("wh","who, what, when")
         do write(i);
1
6
12
Failure
```

5 Goal-Directed Evaluation

- Expression evaluation in Icon is goal-directed; you always try to make every expression succeed and return a value, if at all possible.
- In the example below, find first returns 1. This makes ((i :=...) > 10) fail. Next find generates 14 which makes ((i :=...) > 10) succeed, and write is executed.

```
S := "where and at what time?"
if (i := find("wh", S) > 10) then
  write(i)
```

6 Goal-Directed Evaluation...

```
][ 10 < (1 to 12);
   r34 := 11
][ every write( 10 < (1 to 12));
11
12
Failure
```

7 Counting Vowels

```
procedure main()
   v := 0
   while line := read() do {
      every c := !line do
         if c == !"aeiouAEIOU" then
            v +:= 1
   }
   write("vowels=",v)
end

> vowels
hi there
vowels=3
```

8 `find...`

- The expression

  ```
  S := "where and at what time?"
  ][ every i := 10<find("wh",S) do write(i);
  14
  ```

  can also be written

```
][ every write(10 < find("wh",S));
14
```

9 File Operations

- The following statement copies a file `f1` row-by-row to another file `f2`:

```
while write(f2, read(f1))
```

- Note that `read` is not a generator — hence the use of `while` rather than `every`.

10 Bang!

- `!S` Generates all the characters from the string `S`, or all the elements of the list/table/set `S`.
- `every write(!S)` writes all the characters from the string`S`, one character per line.
- If `S` is a variable then `!S` will generate variables that can be assigned to.

11 Backtracking

- `&fail` always fails.

```
][ &fail;
Failure
][ 3;
   r38 := 3  (integer)
][ 3 + &fail;
Failure
][ 3 + numeric("pi");
Failure
```

12 Bang! — Examples

- Different ways to write the elements of a list:

```
][ L := [1,2,3];
][ every i := !L do write(i);
1
2
3
```

```
][ every write(!L);
1
2
3
```

13 Bang! — Examples...

```
][ every write(L[1 to 3]);
1
2
3
][ write(!L) + &fail;
1
2
3
```

14 Bang!...

- If `L` in `!L` generates variables, then they can be assigned to:

```
][ every !L := 5;
][ L;
   r16 := L1:[5,5,5]
][ !L := 1;
][ L;
   r24 := L1:[1,5,5]
```

15 Bang!...

- Note that literal strings cannot be assigned to:

```
][ S := "bye";
][ write(!S);
b
][ every write(!S);
b
y
e
Failure
][ every !S := "m";
][ S;
   r30 := "mmm"  (string)
][ every !"bye" := "m";
Run-time error 111
```

16 Other Built-In Generators

`?S` Generates random elements from the set, string, table, etc.`S`.

`upto(C, S)` Generate all the positions in the string `S`, where the characters in`C` occur. `C` is a special construction called a`CSet`, a set of characters.`CSets` are written in single quotes, strings in doubles.

```
              12345678901234
upto ('xyz', "zebra-ox-young")
    generates {1, 8, 10}
```

Alternation

17 Alternation

- `expr1 | expr2` generates the values from `expr1`, then from`expr2`.
- `1 | 2 | 3` is the same as `1 to 3`.
- `(1 to 3) | (4 to 6)` is the same as `1 to 6`.
- `&fail | 3` generates `3`.
- `(1=2) | 3` generates `3`.
- `(1=1) | 3` generates `1,3` (since `1=1` succeeds and produces `1`).

18 Variable generation

- The expression `x | y` generates the *variables* `x` and `y`.
- The expression `every (x | y) := 0` is equivalent to `x := 0; y := 0`

19 Terminating Execution

- The built-in procedure `stop(s)` writes `s` and terminates execution.
- A common idiom is `x := p() | stop("error")`. If `p()` fails, then stop and write `"error"`, otherwise assign the result of `p()` to `x`.

20 Variable generation

`every i := (0 | 1) do write (i)` First write `0` then `1`.

`every (x | y) := 0 x := 0; y := 0`

```
][ x := 1;
][ y:= 2;
][ every write(x|y);
1
2
][  every (x|y) := 42;
][ every write(x|y);
42
42
```

21 Examples

```
][ every write(1 | 2 | !"45" | 6);
1
2
4
5
6
][ write((1 | 2 | 3) > 2);
2
   r13 := 2
][ write(2 < (1 | 2 | 3));
3
   r14 := 3
```

22 Examples

```
][ x := 5;
   r16 := 5
][ y := 6;
   r19 := 6
][ (x | y) = 6;
   r20 := 6
```

Procedures as Generators

23 Procedures as Generators

Procedures are really generators; they can return 0, 1, or a sequence of results. There are three cases

`fail` The procedure fails and generates no value.

`return e` The procedure generates one value, `e`.

`suspend e` The procedure generates the value `e`, and makes itself ready to possibly generate more values.

24 Example

```
procedure To(i,j)
   while i <= j do {
      suspend i
      i+:= 1
   }
end

procedure main()
   every k := To(1,3) do
      write(k)
end
```

25 simple.icn

```
procedure P()
   suspend 3
   suspend 4
   suspend 5
end

procedure main()
   every k := P() do
      write(k)
end
```

26 simple.icn. . .

```
> setenv TRACE 100
> simple
             :       main()
simple.icn   :    8  | P()
simple.icn   :    2  | P suspended 3
3
simple.icn   :    9  | P resumed
```

```
simple.icn   :    3  | P suspended 4
4
simple.icn   :    9  | P resumed
simple.icn   :    4  | P suspended 5
5
simple.icn   :    9  | P resumed
simple.icn   :    5  | P failed
simple.icn   :   10  main failed
```

27 simple.icn...

- Remember *goal-directed evaluation* — Icon will resume a generator as many times as necessary in order to try to make an expression succeed.
- The number of times a generator is invoked also depends on the context.

28 simple.icn...

```
][ .inc simple.icn;
][ P();
   r1 := 3  (integer)
][ every write(P());
3
4
5
][ P()=4;
   r3 := 4
][ P() + 10;
   r4 := 13
```

Bounded Expressions

29 Bounded Expressions

- Unlike Prolog, backtracking in Icon is *bounded*. This means that a generator that appears in certain parts of certain control constructs will never generate more than one value.
- `if e1 then e2 else e3` — `e1` is bounded, `e2` and `e3` are not.
- `while e1 do e2` — `e1` and `e2` are both bounded.
- `every e1 do e2` — `e1` is not bounded but `e2` is.
- `{e1, e2, ..., en}` — `e1, e2,...` are bounded but `en` is not.

30 Example

```
][ if write(P()) then &fail else &fail;
3
Failure
][ (if P() then write(P()) else 1) & &fail;
3
4
5
Failure
```

31 Example...

```
][ every i := P() do write(i);
3
4
5
Failure
][ while i := P() do write(i);
3
3
3...
][ {write(P()); 42} & &fail;
3
```

32 Example...

```
][ every i := {write(1 to 5); 42} do write(i);
1
42
][  every i := {write(1 to 5); 10 to 12} do write(i);
1
10
```

```
11
12
][ every i := {write(1 to 5); write(100 to 105); 10 to 12} do write(i);
1
100
10
11
12
```

Summary

33 Readings

- Read `Christopher, pp. 35--42, 44, 56--57`.
- Alternatively, read `Griswold&Griswold, pp. 87--95`.

34 Acknowledgments

- Some material on these slides has been modified from William Mitchell's Icon notes: `http://www.cs.arizona.edu/classes/cs372/fall03/handouts.html`.
- 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`.