Experiences with an Icon-like Expression Evaluation System

Laurence Tratt
http://tratt.net/laurie/

Middlesex University

2010/10/18
The story

1. Find an unusual feature in an ‘old’ language.
The story

1. Find an unusual feature in an ‘old’ language.
2. Try putting it in a ‘new’ language.
The story

1. Find an unusual feature in an ‘old’ language.
2. Try putting it in a ‘new’ language.
3. Fix problems.
The story

1. Find an unusual feature in an ‘old’ language.
2. Try putting it in a ‘new’ language.
3. Fix problems.
Icon history

- Designed by Ralph Griswold (Arizona) in mid/late 70s (v1, late 1978).
- Successor of sorts to SNOBOL4 (via SL5).
- SNOBOL4: essentially a string-matching DSL.
- Icon: a dynamically typed Algol-ish language.
- Very active development until late 80s; (some?) development continuing (v9.5.0 April 2010); runs happily on modern machines.
- Successor languages e.g. Unicon.
Icon history

- Designed by Ralph Griswold (Arizona) in mid/late 70s (v1, late 1978).
- Successor of sorts to SNOBOL4 (via SL5).
- SNOBOL4: essentially a string-matching DSL.
- Icon: a dynamically typed Algol-ish language.
- Very active development until late 80s; (some?) development continuing (v9.5.0 April 2010); runs happily on modern machines.
- Successor languages e.g. Unicon.
- [Personal aside: I ‘found’ Icon through its influence, via Tim Peters, on Python generators.]
Why Icon is interesting

- Programming languages tend to be variations on a theme.
Why Icon is interesting

- Programming languages tend to be variations on a theme.
- Icon explicitly wanted to try new things.
- For its day, several unusual ideas.
Why Icon is interesting

- Programming languages tend to be variations on a theme.
- Icon explicitly wanted to try new things.
- For its day, several unusual ideas.
- Some *still* unusual.

L. Tratt  [http://tratt.net/laurie/](http://tratt.net/laurie/)
Why Icon is interesting

- Programming languages tend to be variations on a theme.
- Icon explicitly wanted to try new things.
- For its day, several unusual ideas.
- Some *still* unusual.
- Case in point: its expression evaluation system.
Why Icon is interesting

- Programming languages tend to be variations on a theme.
- Icon explicitly wanted to try new things.
- For its day, several unusual ideas.
- Some *still* unusual.
- Case in point: its expression evaluation system. *Allows backtracking in an imperative language.*
Icon

- Procedural; dynamically typed; Algol-ish syntax.
Icon

- Procedural; dynamically typed; Algol-ish syntax.
- In 2010, a little ‘old-fashioned’: e.g. differentiating values and references, default values for variables.
- [Not a criticism: we’re all products of our time.]
A little example

Icon version of \texttt{wc -l}:

\begin{verbatim}
procedure main(argv)
  f := open(argv[1], "rt")
  i := 0
  while read(f) do {
    i := i + 1
  }
  write(i)
end
\end{verbatim}

All fairly standard...
Icon version of `wc -l`:

```icon
class main;
    procedure main(argv)
        f := open(argv[1], "rt")
        i := 0
        while read(f) do {
            i := i + 1
        }
        write(i)
    end
end
```

All fairly standard... except the `read` function.
Success and failure

- Standard language: expressions produce values.
Success and failure

- Standard language: expressions produce values.
- Icon expressions:
  - which *succeed* produce values
  - which *fail* do not produce a value and transmit failure to their container.

Note: failure is not like throwing an exception. Exception Something unexpected (probably bad) happened. Failure An expression can produce no more values.
Success and failure

- Standard language: expressions produce values.
- Icon expressions:
  - which *succeed* produce values
  - which *fail* do not produce a value and transmit failure to their container.
- Note: failure is *not* like throwing an exception.
  - Exception Something unexpected (probably bad) happened.
  - Failure An expression can produce no more values.
Success and failure

- Standard language: expressions produce values.
- Icon expressions:
  - which *succeed* produce values
  - which *fail* do not produce a value and transmit failure to their container.

Note: failure is *not* like throwing an exception.
- **Exception** Something unexpected (probably bad) happened.
- **Failure** An expression can produce no more values.

Orthogonal concepts: both can appear in a language.
Success and failure

- Standard language: expressions produce values.
- Icon expressions:
  - which *succeed* produce values
  - which *fail* do not produce a value and transmit failure to their container.

- Note: failure is *not* like throwing an exception.
  - **Exception** Something unexpected (probably bad) happened.
  - **Failure** An expression can produce no more values.

- Orthogonal concepts: both can appear in a language.
- Success / failure are run-time concepts.
Success / failure and boolean logic

- Consider $x < y$:

  - Succeeds (and produces 3) if $x$ is 2 and $y$ is 3.
  - Fails if $x$ is 2 and $y$ is 1.

Icon has no standard boolean logic; no boolean datatype; no boolean operators.

Yet 'standard' code works as expected:

```plaintext
if x < y then {
    write(x)
}
```
Success / failure and boolean logic

Consider \( x < y \):
- succeeds (and produces 3) if \( x \) is 2 and \( y \) is 3.
Success / failure and boolean logic

- Consider $x < y$:
  - succeeds (and produces 3) if $x$ is 2 and $y$ is 3.
  - fails if $x$ is 2 and $y$ is 1.
Success / failure and boolean logic

- Consider $x < y$:
  - succeeds (and produces 3) if $x$ is 2 and $y$ is 3.
  - fails if $x$ is 2 and $y$ is 1.

- Icon has no standard boolean logic; no boolean datatype; no boolean operators.
Success / failure and boolean logic

- Consider $x < y$:
  - succeeds (and produces 3) if $x$ is 2 and $y$ is 3.
  - fails if $x$ is 2 and $y$ is 1.

- Icon has no standard boolean logic; no boolean datatype; no boolean operators.

- Yet ‘standard’ code works as expected:
  
  ```plaintext
  if x < y then {
      write(x)
  }
  ```
Generators

- Icon functions conventionally split into:
  - Procedures generate exactly one value.
Generators

- Icon functions conventionally split into:
  - Procedures generate exactly one value.
  - Generators generate zero or more values.

Example generator:

```plaintext
procedure ito(x)
    i := 0
    while i < x do {
        suspend i
        i := i + 1
    }
end

procedure main()
    every x := ito(10) do { write(x) }
end
```

`suspend` is like Python's `yield`.
`every` is similar to `for`: it pumps a generator to produce all its values. Once the generator fails, `every` fails too.

`c.f. while`: while evaluates its expression anew on every iteration.
Generators

- Icon functions conventionally split into:
  - **Procedures** generate exactly one value.
  - **Generators** generate zero or more values.

- **Example generator:**
  ```icon
  procedure ito(x)
    i := 0
    while i < x do {
      suspend i
      i := i + 1
    }
  end

  procedure main()
    every x := ito(10) do { write(x) }
  end
  ```

- **suspend** is like Python’s **yield**.
- **every** is similar to **for**: it **pumps** a generator to produce all its values.
- Once the generator fails, **every** fails too.
Generators

- Icon functions conventionally split into:
  
  **Procedures** generate exactly one value.
  
  **Generators** generate zero or more values.

- Example generator:

  ```
  procedure ito(x)
  i := 0
  while i < x do {
    suspend i
    i := i + 1
  }
  end

  procedure main()
  every x := ito(10) do { write(x) }
  end
  ```

  **[suspend is like Python’s yield.]**

  **every is similar to for**: it *pumps* a generator to produce all its values.

  Once the generator fails, **every** fails too.

  **c.f. while**: while evaluates its expression anew on every iteration.
Other generators

- \( i \) to \( j \): a built-in \( \text{ito} \).
Other generators

- \( i \) to \( j \): a built-in \( \text{ito} \).
- \textit{Alternation} \( a \mid b \) subsumes boolean OR.
Goal-directed evaluation

A limited form of backtracking.

Conjunction $a \& b$ succeeds iff both $a$ and $b$ succeed.

If $a$ fails, the conjunction fails.

If $b$ fails, $a$ is pumped for a new value and $b$ retried.

Print out the even numbers between 0 and 9 inclusive:

```plaintext
procedure main()
    every x := ito(10) & x % 2 == 0 do {
        write(x)
    }
end
```

Other backtracking features e.g.: reversible assignment $x \leftarrow x$ and limited generation $e \backslash i$.
Goal-directed evaluation

- A limited form of backtracking.
- *Conjunction* $a \land b$ succeeds iff both $a$ and $b$ succeed.

Print out the even numbers between 0 and 9 inclusive:

```plaintext
procedure main()
    every x := ito(10) & x % 2 == 0 do {
        write(x)
    }
end
```

Other backtracking features e.g.: reversible assignment $x \leftarrow x$ and limited generation $e$.
Goal-directed evaluation

- A limited form of backtracking.
- **Conjunction** $a \& b$ succeeds iff both $a$ and $b$ succeed.
- If $a$ fails, the conjunction fails.
- If $b$ fails, $a$ is pumped for a new value and $b$ retried.
- Print out the even numbers between 0 and 9 inclusive:

```plaintext
procedure main()
    every x := ito(10) & x % 2 == 0 do {
        write(x)
    }
end
```
Goal-directed evaluation

- A limited form of backtracking.
- **Conjunction** \( a \& b \) succeeds iff both \( a \) and \( b \) succeed.
- If \( a \) fails, the conjunction fails.
- If \( b \) fails, \( a \) is pumped for a new value and \( b \) retried.

Print out the even numbers between 0 and 9 inclusive:

```
procedure main()
    every x := ito(10) & x % 2 == 0 do {
        write(x)
    }
end
```

- Other backtracking features e.g.: reversible assignment \( x \leftarrow x \) and limited generation \( e \setminus i \).
The extent of backtracking

- Is this like Prolog?

No. Backtracking is local in nature. Chief mechanism: bounded expressions. Roughly: backtracking only occurs within individual lines. Line 2 does not cause backtracking to line 1. A good thing: unlimited backtracking in an imperative language not desirable.
The extent of backtracking

- Is this like Prolog? No.
- Backtracking is local in nature.
- Chief mechanism: bounded expressions.
- Roughly: backtracking only occurs within individual lines.

\[
x := 1 \mid 3 \\
y := x > 2
\]
The extent of backtracking

- Is this like Prolog? No.
- Backtracking is local in nature.
- Chief mechanism: bounded expressions.
- Roughly: backtracking only occurs within individual lines.

```
x := 1 | 3
y := x > 2
```

Line 2 does not cause backtracking to line 1.
The extent of backtracking

- Is this like Prolog? No.
- Backtracking is local in nature.
- Chief mechanism: bounded expressions.
- Roughly: backtracking only occurs within individual lines.

```
x := 1 | 3
y := x > 2
```

Line 2 does not cause backtracking to line 1.

- A good thing: unlimited backtracking in an imperative language not desirable.
Pluses

- Conceptually neat design.
- Backtracking natural for string processing: Icon has special functions for it.
Minuses

- Functions fail by default.

procedure f(x)
  if x > 0 then
    return 1
  end
end

procedure main()
  write(f(-1))
end

prints nothing...

Continual encoding of a boolean datatype.

Generators tend to be hidden.
every f(g(h(...))

Performance issues.

And something else (I'll come back to it).

L. Tratt http://tratt.net/laurie/
Functions fail by default.

```plaintext
procedure f(x)
  if x > 0 then {
    return 1
  }
end

procedure main()
  write(f(-1))
end
```

prints nothing...
Minuses

- Functions fail by default.

```plaintext
procedure f(x)
    if x > 0 then {
        return 1
    }
end

procedure main()
    write(f(-1))
end

prints nothing...

- Continual encoding of a boolean datatype.
Minuses

- Functions fail by default.

```plaintext
procedure f(x)
    if x > 0 then {
        return 1
    }
end

procedure main()
    write(f(-1))
end
```

prints nothing...

- Continual encoding of a boolean datatype.
- Generators tend to be hidden.

```plaintext
every f(g(h(...)))
```
Minuses

- Functions fail by default.

```plaintext
procedure f(x)
    if x > 0 then {
        return 1
    }
end

procedure main()
    write(f(-1))
end
```

prints nothing...

- Continual encoding of a boolean datatype.
- Generators tend to be hidden.

```plaintext
every f(g(h(...)))
```

- Performance issues.
Functions fail by default.

```plaintext
procedure f(x)
    if x > 0 then {
        return 1
    }
end

procedure main()
    write(f(-1))
end
```

prints nothing...

Continual encoding of a boolean datatype.

Generators tend to be hidden.

```plaintext
every f(g(h(...)))
```

Performance issues.

And something else (I’ll come back to it).
Converge

- A ‘modern’ Python-ish language with macros.
- First non-Icon clone with an Icon-like expression evaluation system.
- Initially slurped in wholesale from Icon...
- ...then tweaked over time.
- More at http://convergepl.org/
Fix #1

- Recap: functions fail by default.
Fix #1

- Recap: functions fail by default.
- Functions return null by default.
- Must explicitly use (equivalent of) return fail.
- Debugging suddenly much much easier.
Recap: continual encoding of a boolean datatype.

Lack of a boolean datatype a real irritant.

Is there an Icon-esque solution?

Introduce fail singleton object. If evaluated in e.g. an if conditional, causes failure.

Ta-da! Works well for all common cases.

Except...

fail is a top-level variable in every module. Module can return the value associated with a var.

\[ x := \text{mod.get_var("fail")} \]

where mod_var does return fail, so no assignment is made to \( x \).

I lost two days debugging this one. Unfortunate conclusion: it doesn’t really work.
(Attempted) fix #2

- Recap: continual encoding of a boolean datatype.
- Lack of a boolean datatype a real irritant.
- Is there an Icon-esque solution?
Recap: continual encoding of a boolean datatype.
Lack of a boolean datatype a real irritant.
Is there an Icon-esque solution?
Introduce \texttt{fail} singleton object.
If evaluated in e.g. an \texttt{if} conditional, causes failure.

Ta-da! Works well for all common cases.

Except...

\texttt{fail} is a top-level variable in every module.
Module can return the value associated with a var.
\texttt{x := mod.get_var("fail")}
where \texttt{mod_var} does return \texttt{fail}, so no assignment is made to \texttt{x}.

I lost two days debugging this one. Unfortunate conclusion: it doesn't really work.
(Attempted) fix #2

- Recap: continual encoding of a boolean datatype.
- Lack of a boolean datatype a real irritant.
- Is there an Icon-esque solution?
- Introduce `fail` singleton object.
- If evaluated in e.g. an `if` conditional, causes failure.
- Ta-da! Works well for all common cases.

Except...

```plaintext
fail is a top-level variable in every module.
Module can return the value associated with a var.
x := mod.get_var("fail")
where `mod_var` does return `fail`, so no assignment is made to `x`.
I lost two days debugging this one. Unfortunate conclusion: it doesn't really work.
```
Recap: continual encoding of a boolean datatype.
Lack of a boolean datatype a real irritant.
Is there an Icon-esque solution?
Introduce `fail` singleton object.
If evaluated in e.g. an `if` conditional, causes failure.
Ta-da! Works well for all common cases.
Except...
(Attempted) fix #2

- Recap: continual encoding of a boolean datatype.
- Lack of a boolean datatype a real irritant.
- Is there an Icon-esque solution?
- Introduce `fail` singleton object.
- If evaluated in e.g. an `if` conditional, causes failure.
- Ta-da! Works well for all common cases.
- *Except...* `fail` is a top-level variable in every module.
- Module can return the value associated with a var.
Recap: continual encoding of a boolean datatype.
Lack of a boolean datatype a real irritant.
Is there an Icon-esque solution?
Introduce `fail` singleton object.
If evaluated in e.g. an `if` conditional, causes failure.
Ta-da! Works well for all common cases.
Except... `fail` is a top-level variable in every module.
Module can return the value associated with a var.
```plaintext
x := mod.get_var("fail")  
where mod_var does return  
fail, so no assignment is made to x.
```

(Attempted) fix #2

- Recap: continual encoding of a boolean datatype.
- Lack of a boolean datatype a real irritant.
- Is there an Icon-esque solution?
- Introduce `fail` singleton object.

If evaluated in e.g. an `if` conditional, causes failure.

Ta-da! Works well for all common cases.

Except... `fail` is a top-level variable in every module.

Module can return the value associated with a var.

```plaintext
x := mod.get_var("fail") where mod_var does return fail, so no assignment is made to x.
```

I lost two days debugging this one. Unfortunate conclusion: it doesn’t really work.
Fix #3

- Recap: generators are hidden.

Iter_
Fix #3

- Recap: generators are hidden.
- Fix: conventionally prefix all generator names with `iter_`.
- Simple and effective.
Item #4

- Recap: performance issues.
Recap: performance issues.
Icon and Converge stack-based VMs.
Goal-directed evaluation requires *huge* numbers of stack operations.
The only optimised part of the Converge VM and *still* very slow.
Recap: performance issues.

Icon and Converge stack-based VMs.

Goal-directed evaluation requires *huge* numbers of stack operations.

The only optimised part of the Converge VM and *still* very slow.

Icon seems to require a stack-based VM.
Recap: performance issues.
Icon and Converge stack-based VMs.
Goal-directed evaluation requires *huge* numbers of stack operations.
The only optimised part of the Converge VM and *still* very slow.
Icon seems to require a stack-based VM. Or does it?
Full paper has suggestions for an efficient register-based VM.
Experiences (bad)

- The bad news: Converge users don’t use most of the Icon features.
- Explanation #1: too stuck in our ways.
Experiences (bad)

- The bad news: Converge users don’t use most of the Icon features.
- Explanation #1: too stuck in our ways.
- Explanation #2: backtracking great for string processing. But we have regular expressions and formal parsing systems.

In Icon:
```
sentence ? while tab(upto(letters)) do write(tab(many(letters)))
```
is (in Python) roughly:
```
print re.split(\s+, sentence)
```

Explanation #3: backtracking isn’t expressive enough. Icon’s backtracking can’t (shouldn’t!) match Prolog’s; inevitably less expressive.

My conclusion: for normal modern programming, goal-directed evaluation isn’t that useful.
The bad news: Converge users don’t use most of the Icon features.

Explanation #1: too stuck in our ways.

Explanation #2: backtracking great for string processing. But we have regular expressions and formal parsing systems. In Icon:

```icon
sentence ? while tab(upto(letters)) do
  write(tab(many(letters))

is (in Python) roughly:
print re.split("\\s+", sentence)
```
The bad news: Converge users don’t use most of the Icon features.

Explanation #1: too stuck in our ways.

Explanation #2: backtracking great for string processing. But we have regular expressions and formal parsing systems.

In Icon:

```icon
sentence ? while tab(upto(letters)) do
  write(tab(many(letters))
```

is (in Python) roughly:

```python
print re.split("\s+", sentence)
```

Explanation #3: backtracking isn’t expressive enough. Icon’s backtracking can’t (shouldn’t!) match Prolog’s; inevitably less expressive.
The bad news: Converge users don’t use most of the Icon features.

Explanation #1: too stuck in our ways.

Explanation #2: backtracking great for string processing. But we have regular expressions and formal parsing systems. In Icon:

```icon
sentence ? while tab(upto(letters)) do
    write(tab(many(letters))
```

is (in Python) roughly:

```python
print re.split("\\s+", sentence)
```

Explanation #3: backtracking isn’t expressive enough. Icon’s backtracking can’t (shouldn’t!) match Prolog’s; inevitably less expressive.

My conclusion: for normal modern programming, goal-directed evaluation isn’t that useful.
Generators are great (we all knew that).
Generators are great (we all knew that).

Failure is a natural idiom.

Consider this common idiom ‘print an item \(x\) if it’s in the dict’:

```python
    d := Dict{"a" : 2, "b" : 8}
    if d.contains("a"):
        Sys::println(d.get("a"))
```

Note duplicated lookup: slow and maintenance nightmare.

Not uncommon to see:

```python
    d := Dict{"a" : 2, "b" : 8}
    try:
        v := d.get("j")
        Sys::println(v)
    catch Exceptions::Key_Exception:
        pass
```

Eugh!
Generators are great (we all knew that).

Failure is a natural idiom.

Consider this common idiom ‘print an item \( x \) if it’s in the dict’:

```cpp
    d := Dict{"a" : 2, "b" : 8}
    if d.contains("a"):
        Sys::println(d.get("a"))
```

Note duplicated lookup: slow and maintenance nightmare.

Not uncommon to see:

```cpp
    d := Dict{"a" : 2, "b" : 8}
    try:
        v := d.get("j")
        Sys::println(v)
    catch Exceptions::Key_Exception:
        pass
```
Generators are great (we all knew that).

Failure is a natural idiom.

Consider this common idiom ‘print an item $x$ if it’s in the dict’:

```python
d := Dict{"a" : 2, "b" : 8}
if d.contains("a"):
    Sys::println(d.get("a"))
```

Note duplicated lookup: slow and maintenance nightmare.

Not uncommon to see:

```python
d := Dict{"a" : 2, "b" : 8}
try:
    v := d.get("j")
    Sys::println(v)
    catch Exceptions::Key_Exception:
        pass
```

Eugh!
In Converge:

```cpp
if x := d.find("a"):
    Sys::println(x)
```

The idiom:

- `find(x)` succeeds if `x` is found; fails otherwise.
- `get(x)` throws an exception if `x` is not found.

A beautiful idiom: used throughout the Converge libraries.
Experiences (good) (cont.)

- In Converge:
  ```
  if x := d.find("a"):
      Sys::println(x)
  ```

- The idiom:
  - `find(x)` succeeds if `x` is found; fails otherwise.
  - `get(x)` throws an exception if `x` is not found.

- A beautiful idiom: used throughout the Converge libraries.

- Failure in `ifs`, in general, is great.
Icon’s expression evaluation system is unique, brilliantly designed, and clever.
Icon’s expression evaluation system is unique, brilliantly designed, and clever.

Useful back in the day; less so now (but perhaps for DSLs?).
Icon’s expression evaluation system is unique, brilliantly designed, and clever.

Useful back in the day; less so now (but perhaps for DSLs?).

But failure in ifs is a thing of beauty.
Summary

- Icon’s expression evaluation system is unique, brilliantly designed, and clever.
- Useful back in the day; less so now (but perhaps for DSLs?).
- But failure in `ifs` is a thing of beauty.
- Open question: does failure in `ifs` require an Icon-like approach? Would it fit into other languages?
Final thoughts

- It seems like a mixed message. But I’m glad I tried.
Final thoughts

- It seems like a mixed message. But I’m glad I tried.
- Icon a great example of a language which defies conventions.
- I wish there were more languages that took that route!
Final thoughts

- It seems like a mixed message. But I’m glad I tried.
- Icon a great example of a language which defies conventions.
- I wish there were more languages that took that route!

Thanks for listening