Lessons learned from an unusual language

Dennis Furey, Ph.D.

Faculty of Business, Computing, and Information Management
London South Bank University

February 12, 2009

anything new under the sun?

Three function definitions in a language called Ursala

```
iprod  = plus:-0.+ times*p
mmult  = iprod*rlD*rK7lD
eudist = sqrt+ iprod+ ~&iiX+ minus*p
```

for three familiar vector operations

- inner product
- matrix multiplication
- Euclidean distance

salient features

- functional style
- single assignment
- no variables or loop counters
- no explicit function parameters required
- no type declarations required
- operators galore

what operators?

Old standards copied from Squiggol and FP

- functional composition *f*+*f*
- map over a list *f*`*`
- reduce over a list *f*`:-`*a*
- map over a pair *f*`~~`
- lots more

and new ones made to order on the fly

- duplicate `~&iiX`
- map over the zip of a pair of lists (zipwith) *f*`*`p
- infinitely more

what else?

- numerical libraries
- arbitrary precision arithmetic
- 3-D graphics
- object-like smart records
- polymorphism
- financial derivatives data structures
- client/server interaction
- free open source license

Outline

Outline

Outline

generalized identity functions

- deconstruct a pair
  - `~&l (x,y)` = `x`
  - `~&r (x,y)` = `y`
- deconstruct a list
  - `~&h <u,v,w>` = `u`
  - `~&t <u,v,w>` = `<v,w>`
- deconstruct a pair of lists
  - `~&lt (<a,b,c>,<u,v,w>)` = `<b,c>`
  - `~&rh (<a,b,c>,<u,v,w>)` = `u`
- or a list of pairs
  - `~&thr <(a,b),(c,d),(e,f)>` = `d`
  - `~&tthl <(a,b),(c,d),(e,f)>)` = `e`

generalized a little more

- atomically deconstruct and reconstruct a pair
  `~&rlX (x,y)` = `(y,x)`
- do it to a pair of lists
  `~&rhPltPC (<a,b,c>,<u,v,w>)` = `<u,b,c>`

can’t resist a little more

flip and zip!

```
~&rlxPp (’abc’,<1,2,3>) = <(1,‘c),(2,‘b),(3,‘a)>
```

flatten a tree!

```
~&dvLPCo ‘a^: <‘b^: <‘c^: <>, ‘d^: <>>, ‘e^: <>> = ’abcde’
```

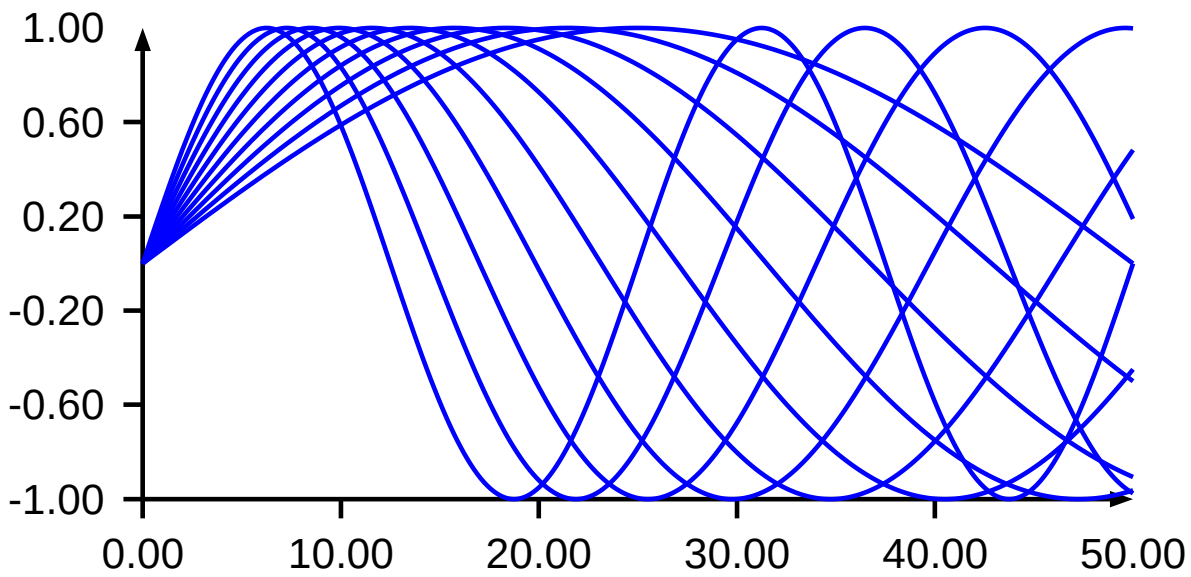
what’s not to like about it?

Outline

functions by the yard

“give me a list of *n* sinusoidal functions with wavelengths ranging from *s* to *l* in geometric progression”

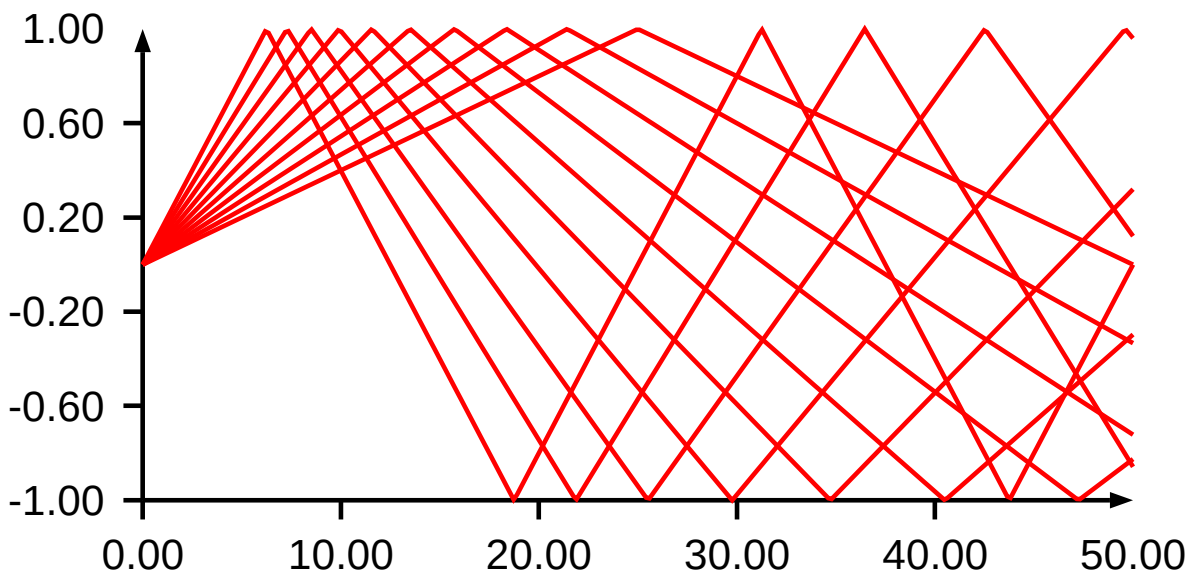
```
(sin_basis "n") ("s","l") =
  (sin++ times/times(2.,pi)++ \/div)* geo"n"/"s" "l"
```



functions by the yard

“make that a saw tooth family”

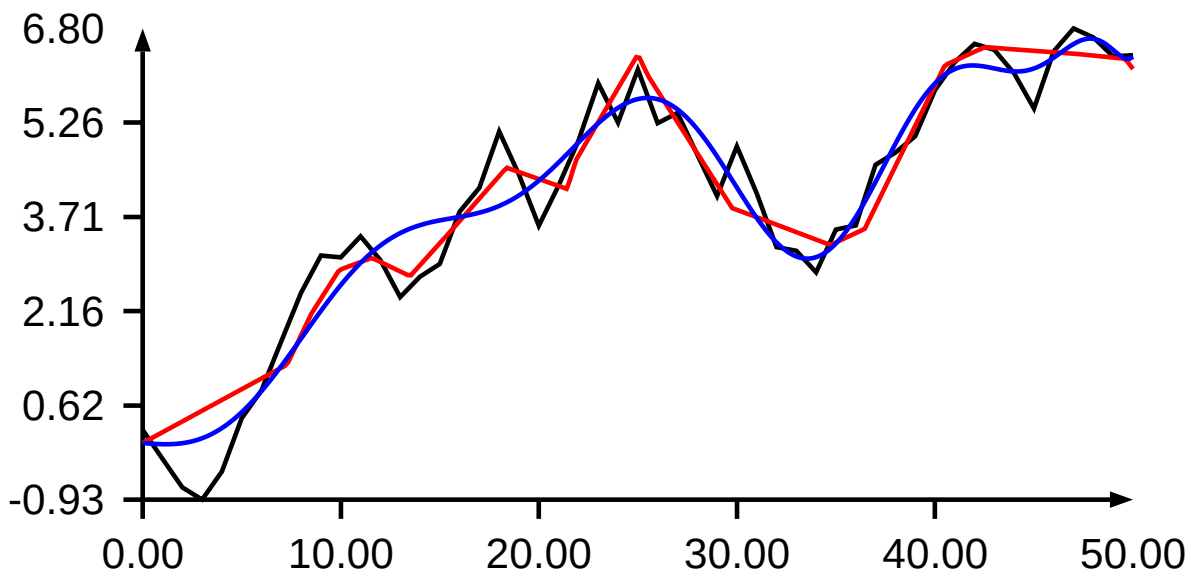
```
saw_basis "n" =
   geo"n"; * -+
      plin<(0.,0.),(0.25,1.),(0.75,-1.),(1.,0.)>+,
      minus^(~&,math..trunc)++ \/div+-
```



curve fitting with Lapack

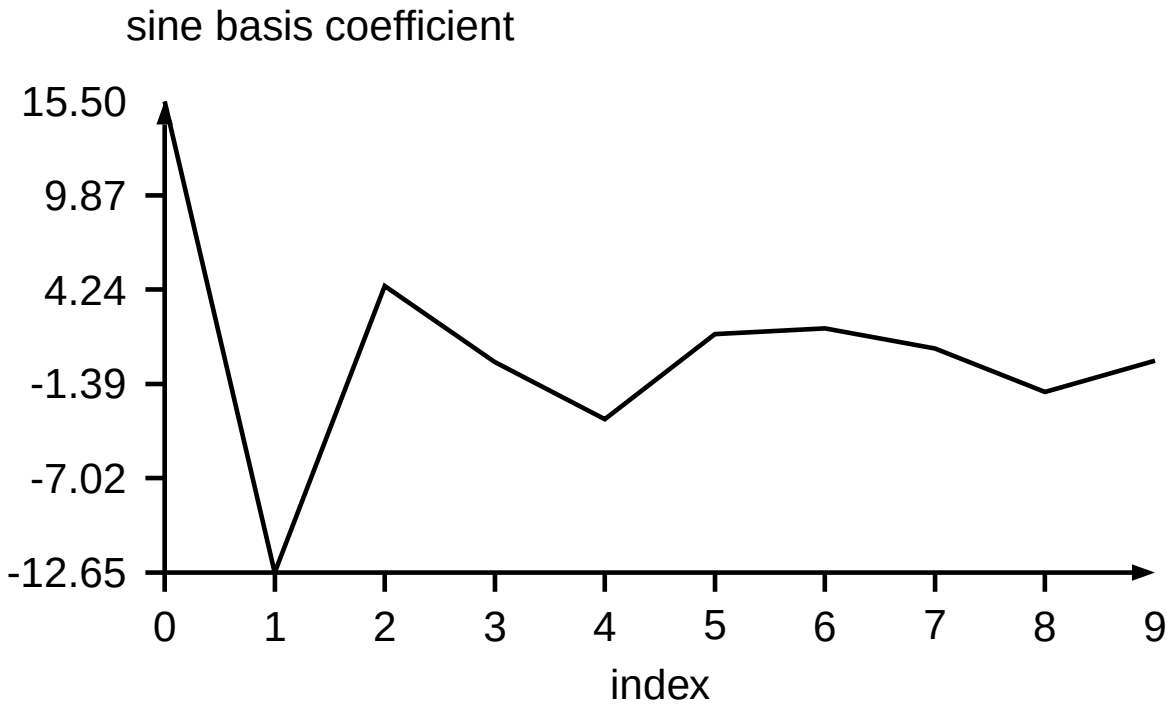
“express a dataset in terms of a basis I might want to change”

```
(regression "b") "x" =
   iprod/(coefficients"b" "x")+ gang "b"

coefficients "b" =
   lapack..dgelsd^\~& gang"b"*+ float*+ iol
```



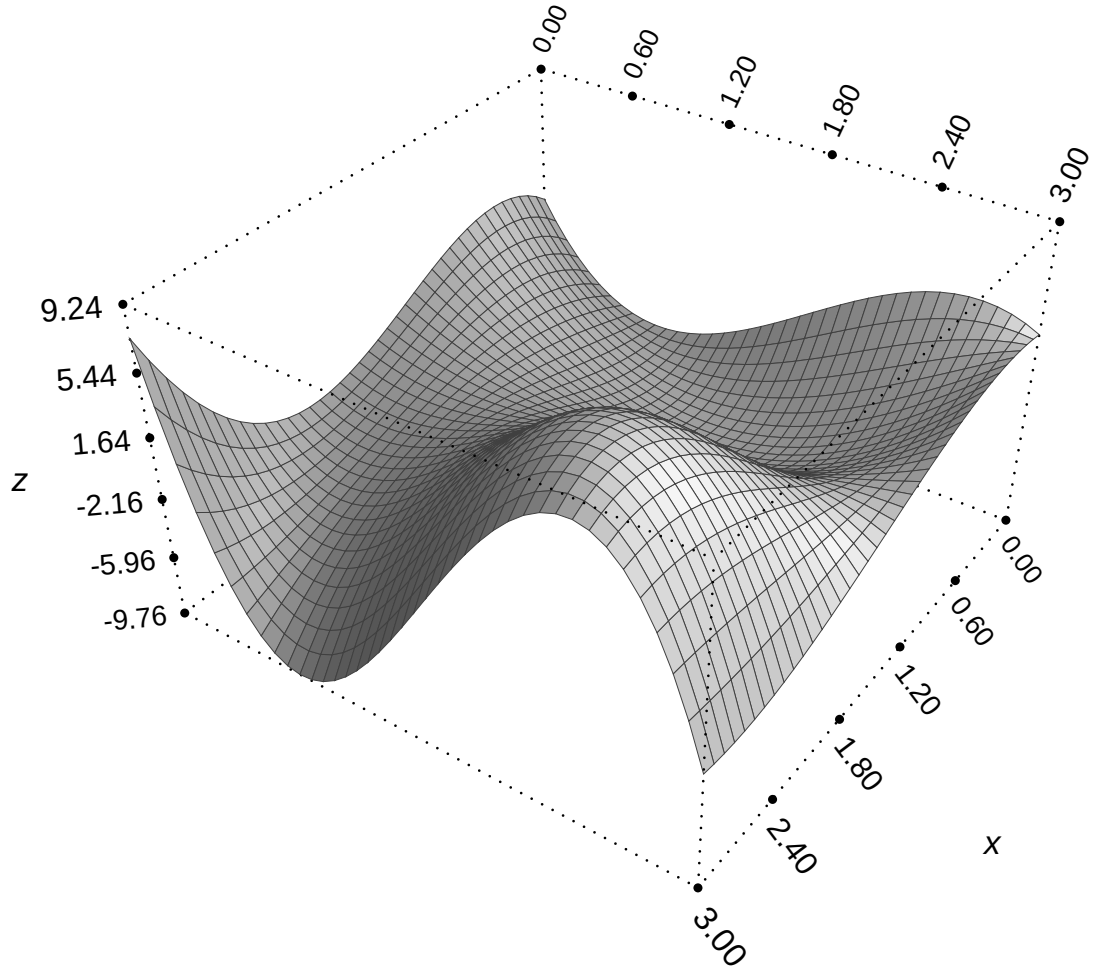
dimensionality reduction

“show me the image in sinusoidal space”



surface rendering with LaTeX

“I’m stuck for a poster presentation.”



Outline

record declaration syntax

A user-defined container of heterogenous types

⟨*record mnemonic*⟩ : :

⟨*field identifier*⟩ ⟨*type expression*⟩ ⟨*initializing function*⟩

⋮

⟨*field identifier*⟩ ⟨*type expression*⟩ ⟨*initializing function*⟩

- fields can be functions or any other type
- fields can be automatically inferred from other fields
- invariants can be automatically maintained
- untyped, opaque, and free union fields are also possible

smart record example

A point record maintains both the polar and rectangular representations, and has a default value of $(0, 0)$.

$$x = r\cos(t) \qquad r = \sqrt{x^2 + y^2}$$
$$y = r\sin(t) \qquad t = \arctan(y/x)$$

smart record example

A point record maintains both the polar and rectangular representations, and has a default value of $(0, 0)$.

```
point ::

x %eZ -|
   -|~x,-&~r,~t,times^/~r cos+ ~t&-|-,
   -|~r,! 0.|-|-
y %eZ -|~y,-&~r,~t,times^/~r sin+ ~t&-,! 0.|-
r %eZ -|
   -|~r,-&~x,~y,sqrt+ plus+ sqr^~/~x ~y&-|-
   -|~x,~y,! 0.|-|-
t %eZ -|
   -|~t,-&~x,~y,math..atan2^/~y ~x&-|-,
   -|~y&& ! div\2. pi,! 0.|-|-
```

parameterized records

An example of a polymorphic set parameterized by the base type with the cardinality automatically maintained:

```
polyset "t" ::
  elements   "t"%S
  cardinality %n  length+ ~elements
```

usage examples:

```
realset = polyset %e

x = realset[elements: {1.0,2.0}]

y = (polyset %s)[elements: {'foo','bar'}]
```

parameterized records

Another example, a rose tree parameterized by a pair of types, with node types alternating by level:

```
rose_tree ("x","y") ::
  root     "x"
  subtrees (_rose_tree ("y","x"))%L
```

- Instantiate as `rose_tree(%n,%e)` for a type of tree with a natural number in the root, IEEE double precision numbers on the next level, naturals again below, and so on.
- Each node has a list of descendents.

Outline

circular definitions

Suppose a function is decared in the form

$$f = d(f)$$

like this list reversal function, for example

```
rev = ~&i&& ("h":"t"). (rev "t")--<"h">
```

but the compiler doesn't understand circular definitions.

What to do?

fixed point combinators

Use a fixed point combinator!

- $f = d(f)$ means f is a fixed point of d
- Plug d into a fixed point combinator to get f

A fixed point combinator for first order functions is given by

```
fix "d" = refer ("f","a").("d" refer "f") "a"
```

where `(refer "f") "x" = "f" ("f","x")`

For example

```
rev= fix "r". ~&i&& ("h":"t"). "r"("t")--<"h">
```

is a non-circular definition of list reversal.

why does this work?

```
(fix d) x
  = (refer ("f","a"). (d refer "f") "a") x
  = (("f","a"). (d refer "f") "a") (
       ("f","a"). (d refer "f") "a",
       x)
  = (d refer ("f","a"). (d refer "f") "a") x
  = (d fix d) x
```

which implies

```
fix d = d fix d
```

so `fix d` is a fixed point of d

why do we care?

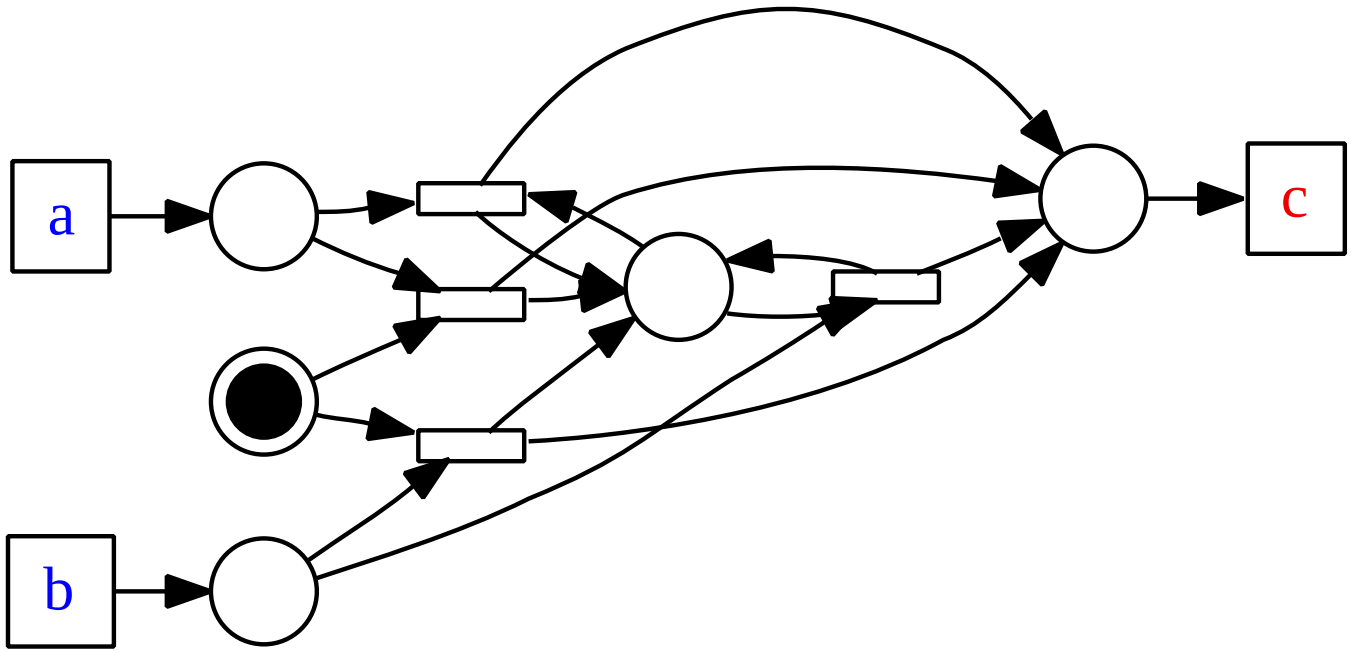
Arbitrary fixed point combinators can be nominated through the `#fix` directive and used by the compiler for solving systems of recurrences.

```
#fix "d". refer ("f","a"). ("d" refer "f") "a"

rev = ~&i&& ("h":"t"). (rev "t")-- <"h">
```

other applications

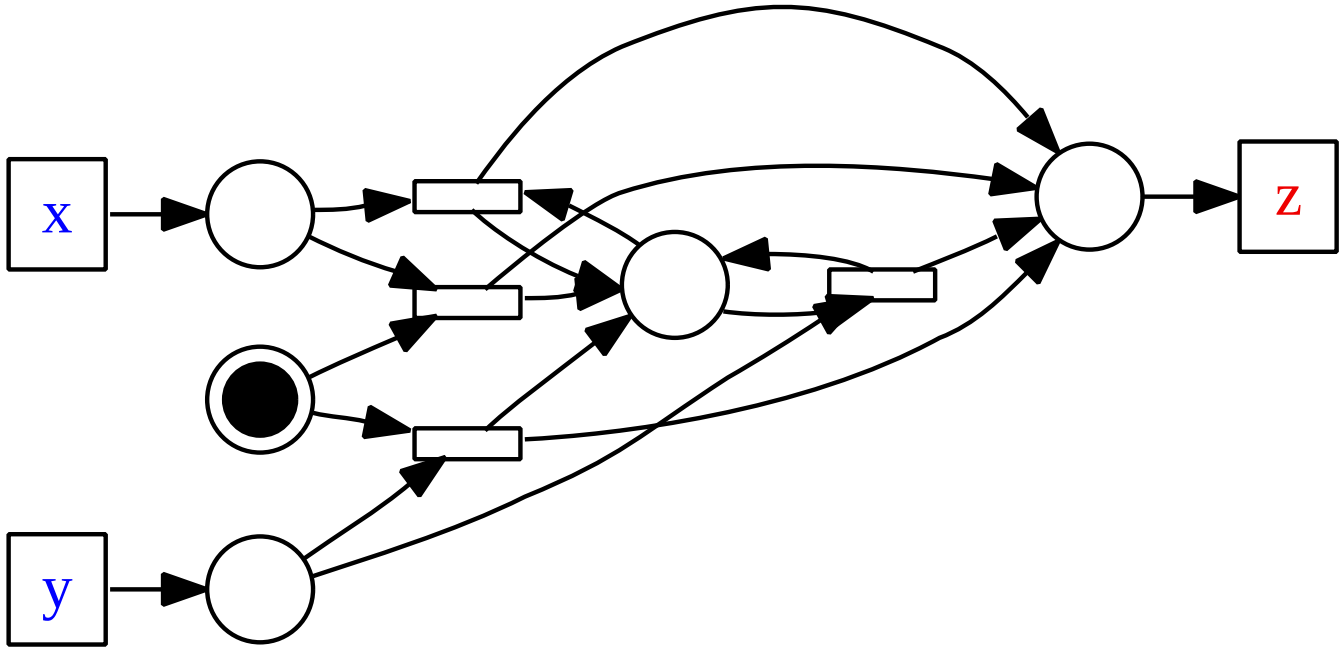
Recursively defined Petri nets

```
#import pnc
#fix pnc-fix

xor = do<getany<'a','b'>,put<'c'>,xor>
```



other applications

```
#fix fix_lifter(1) pnc-fix

xor("a","b","c") =
    do<getany<"a","b">,put<"c">,xor>

net = xor('x','y','z')
```



Outline

Outline

progress or aberration?

As a notation matures, explicit parameters are used less.
(Wolfram, Scott)

```
fix "d"
   = refer ("f","a"). ("d" refer "f") "a"
   = refer ^H("d"+ refer+ ~&f,~&a)
   = refer ^|H("d"+ refer,~&)
   = refer ^|H\~& "d"+ refer
   = refer ^|H\~& refer; "d"

fix = refer+ ^|H\~&+ refer;
```

beautiful or useful?

Recall that if a first order function `f` satisfies

`f = h(f)`

then

`f = fix h = fix "f".  h("f")`

Consider a generalization `gfix` of `fix` such that

`f = "`x_1`"...."`x_n`".  h(f,"`x_1`"..."`x_n`")`

implies

`f = gfix(`n`)"f"."`x_1`"...."`x_n`".  h("f","`x_1`"..."`x_n`")`

beautiful or useful?

Then

```
gfix 0 = fix

gfix 1 =

"h". /// refer ^H(
   ^H("h"+ ///+ refer+ ~&f,~&al),
   ~&ar)

gfix 2 =

"h". /// /// refer ^H(
   ^H(
      ^H("h"+ ///+ ///+ refer+ ~&f,~&all),
      ~&alr),
   ~&ar)
```

beautiful or useful?

Yes, but what's the general case?

(look away now if squeamish)

Three possibilities:

- This notation is so rude that there has to be a better way.
- This idea is so unwelcome we shouldn't care if it's inexpressible.
- Our aesthetic sensibilities need re-examination.

beautiful or useful?

Yes, but what's the general case?

```
gfix =

iota; +^(~&l,+^/~&r ;+ ~&f;+ ~&l)^(
   refer;+ -++-+ * ! ///,
   -++-+ (/*/\ ^H)+ ~&iNX|\NNiXXS+ --<&>+ &r!*)
```

Three possibilities:

- This notation is so rude that there has to be a better way.
- This idea is so unwelcome we shouldn't care if it's inexpressible.
- Our aesthetic sensibilities need re-examination.

Outline

language designers against programmers

Why the relationship is inherently adversarial:

- expressiveness implies more labor saving features
- labor saving features lead to personal dialects
- less work for the writer means more work for the reader
- group productivity drops even if individual productivity rises

The language designer must give priority to the group and the individual must conform. More reasons:

- cultural differences (academic versus commercial)
- axes to grind on both sides

further work

Work on Ursala's expressive power is largely concluded.

Now start at the bottom and meet in the middle:

- know any new machines poking around for a better HLL? (e.g. GPU, cloud, multi-core)
- raise the abstraction without hiding the metal
- scavenge features from Ursala to top it up
- try a significant practical application as a case study

(obvious) conclusions

- Programming languages are cultural artifacts that do not exist in a vacuum.
- The technological problems of language design are secondary.
- New languages can come into general use only when circumstances permit.
- People with strong opinions about programming languages can design their own and use them exclusively.
- Working in this area is necessarily its own reward.