Plumber - A Higher Order Data Flow Visual Programming Language in Lisp

Seika Abe

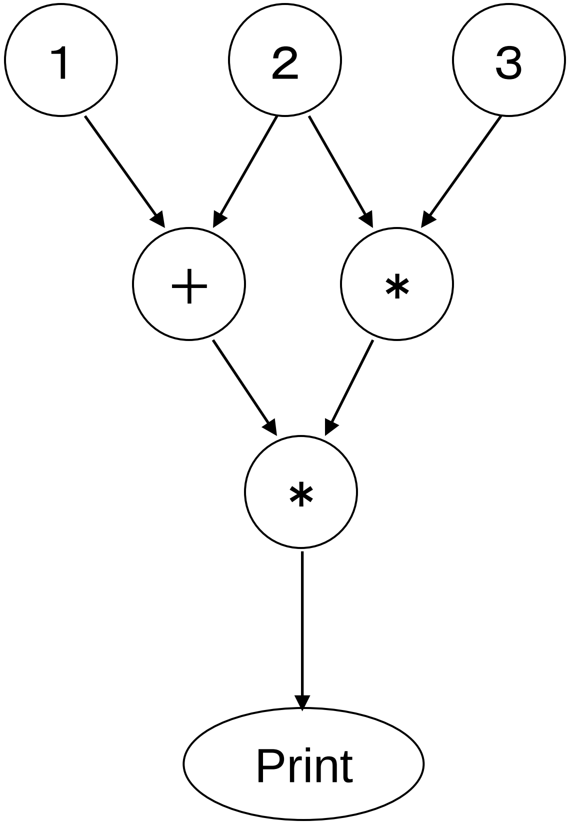
MSI

Outline

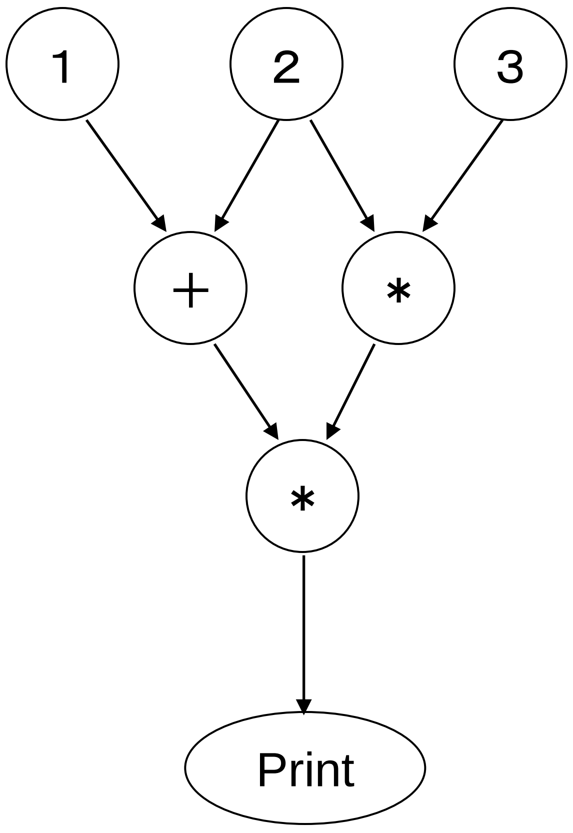
1. What is DFVPL
2. Plumber
3. Streams
4. Higher order functions
5. Finding monads

1/5. What is DFVPL

Dataflow model



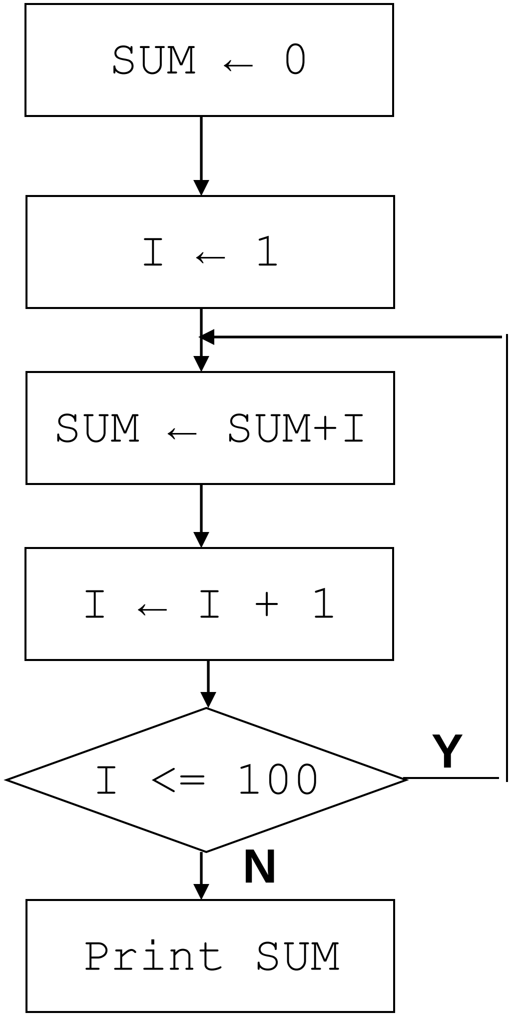
DFVPL is a nice notation



```
load r1,1
load r2,2
add  r1,r1,r2
load r3,3
mul  r2,r2,r3
add  r1,r1,r2
out  r1
```

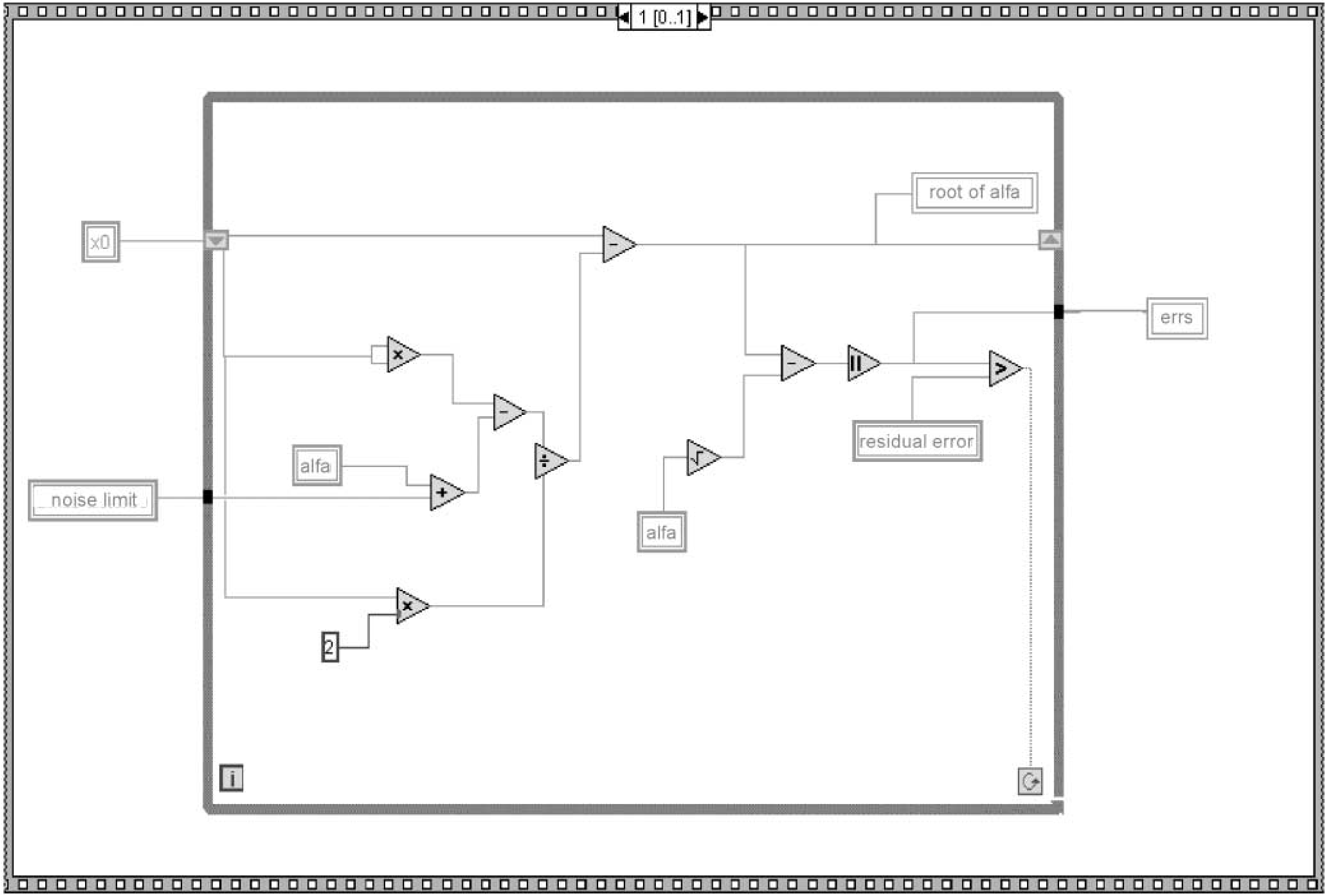
```
(print (* (+ 1 2) (* 2 3)))
```

Flow chart is not a DFVPL

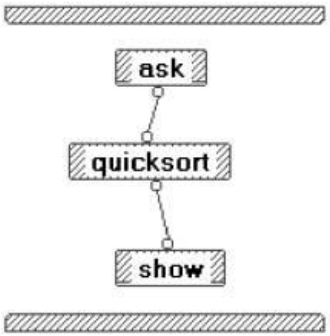


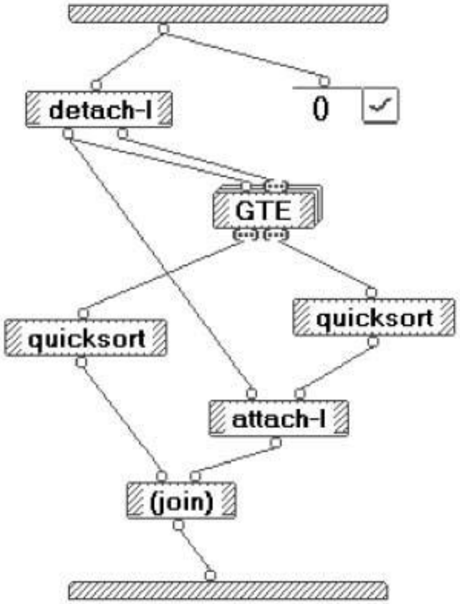
```
SUM := 0
I := 1
do
  SUM := SUM + I
  I := I + 1
while I <= 100
print SUM
```

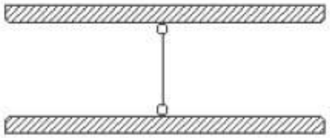
LabVIEW

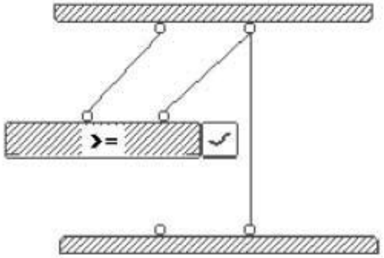

1 [0..1]
x0
noise limit
alfa
2
alfa
root of alfa
errs
residual error

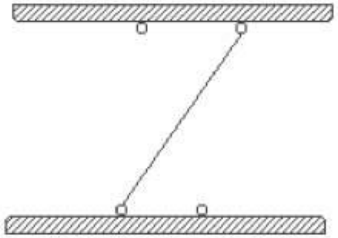
Prograph


Case 1 of method *Qsort*


Case 1 of method *quicksort*


Case 2 of method *quicksort*

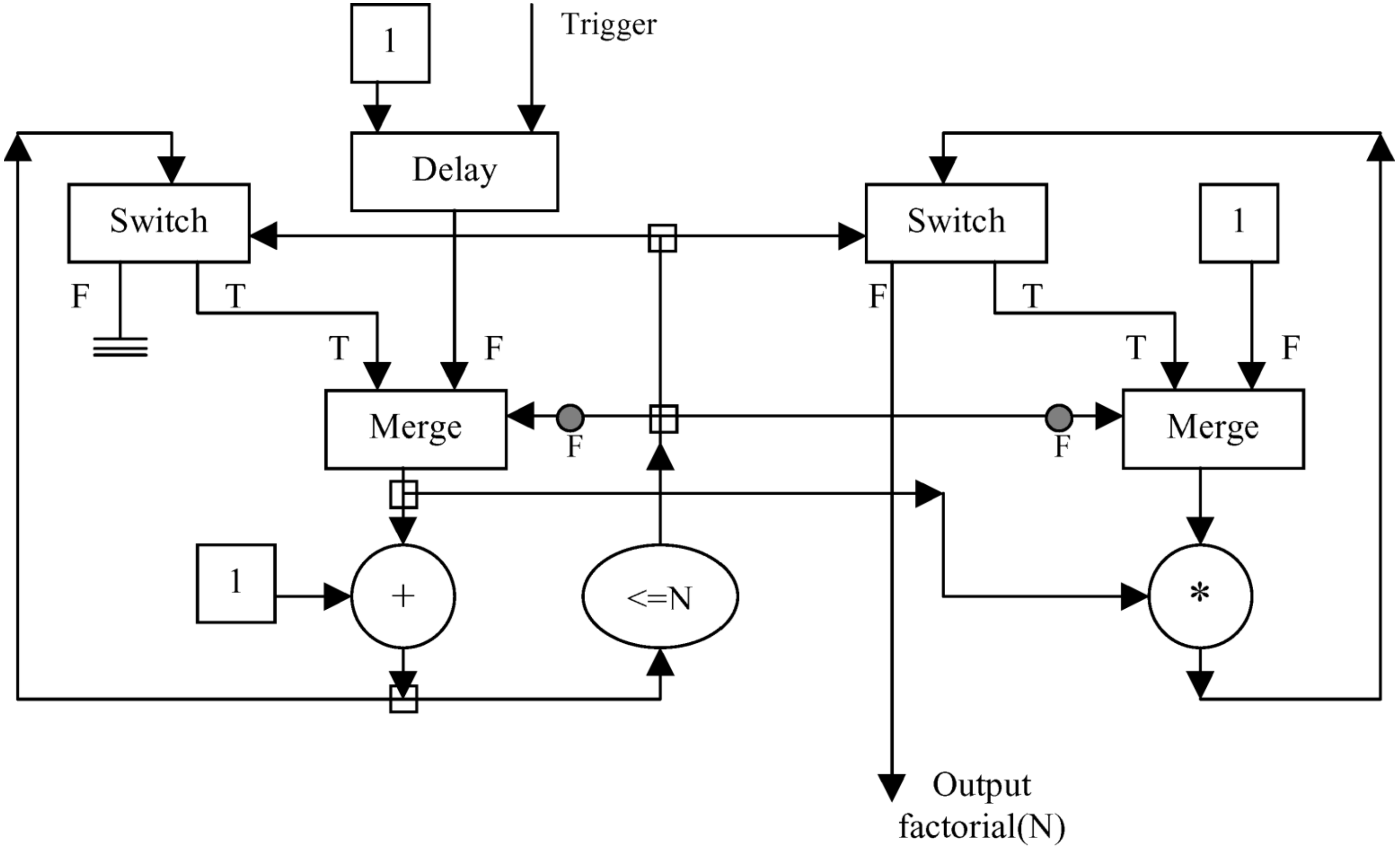

Case 1 of method *GTE*


Case 2 of method *GTE*

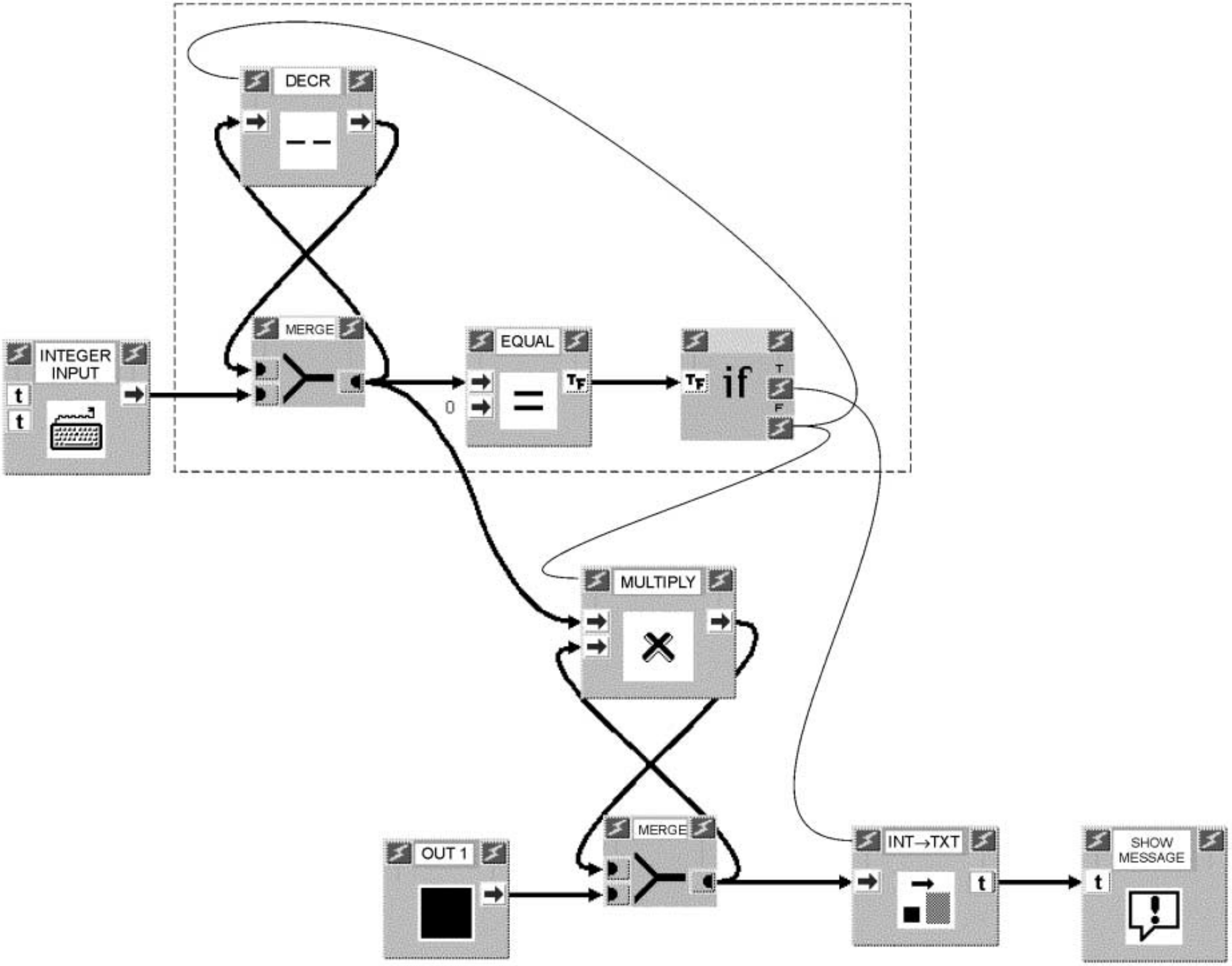
DFVPL in the literature

- Jack B. Dennis, **First version of a data flow procedure language**,1974.
  - Early work of DFVPL
- M. Mosconi and M. Porta, **Iteration constructs in data-flow visual programming languages**, 2000.
  - A study of loop in DFVPL and also a good survey
- W. M. JOHNSTON, **Advances in Dataflow Programming Languages**, 2004.
  - A good survey on DFVPL

Dennis’s factorial in his DFVPL



Mosconi's factorial in his DFVPL



2/5. Plumber

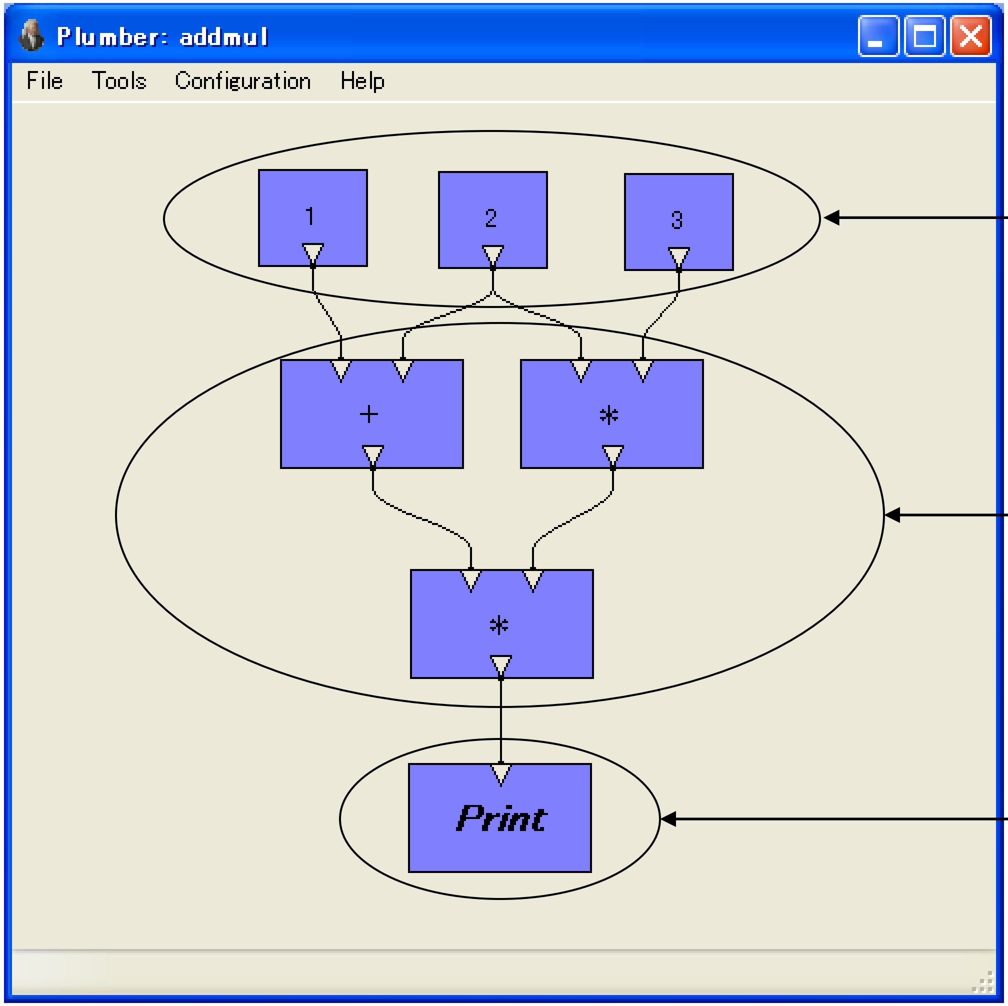
Plumber

- Implemented in Lisp (Allegro CL)
- Consists of A graph editor for DFVPL
- And a compiler from DFVPL to Lisp
- **Nodes can be categorized into two**
- **One is functinal and the other is special**
- **Functional nodes have the dataflow semantics**
- **Functional nodes can be any Lisp functions**
- **Special nodes are built-in and its own semantics**
- Each node corresponds to a thread
- Provides higher-order function feature

Behavior of function nodes

1. Wait until all of inpput data are available
2. Then, make a data list *args* of the all input
3. Apply function of the node to *args*
4. Send the result to the output port
5. N-values result are sent to N oputput port
6. GOTO Step 1.

addmul



constant node

function node

special node

(Shown in ***slant bold***)

Demonstration 1

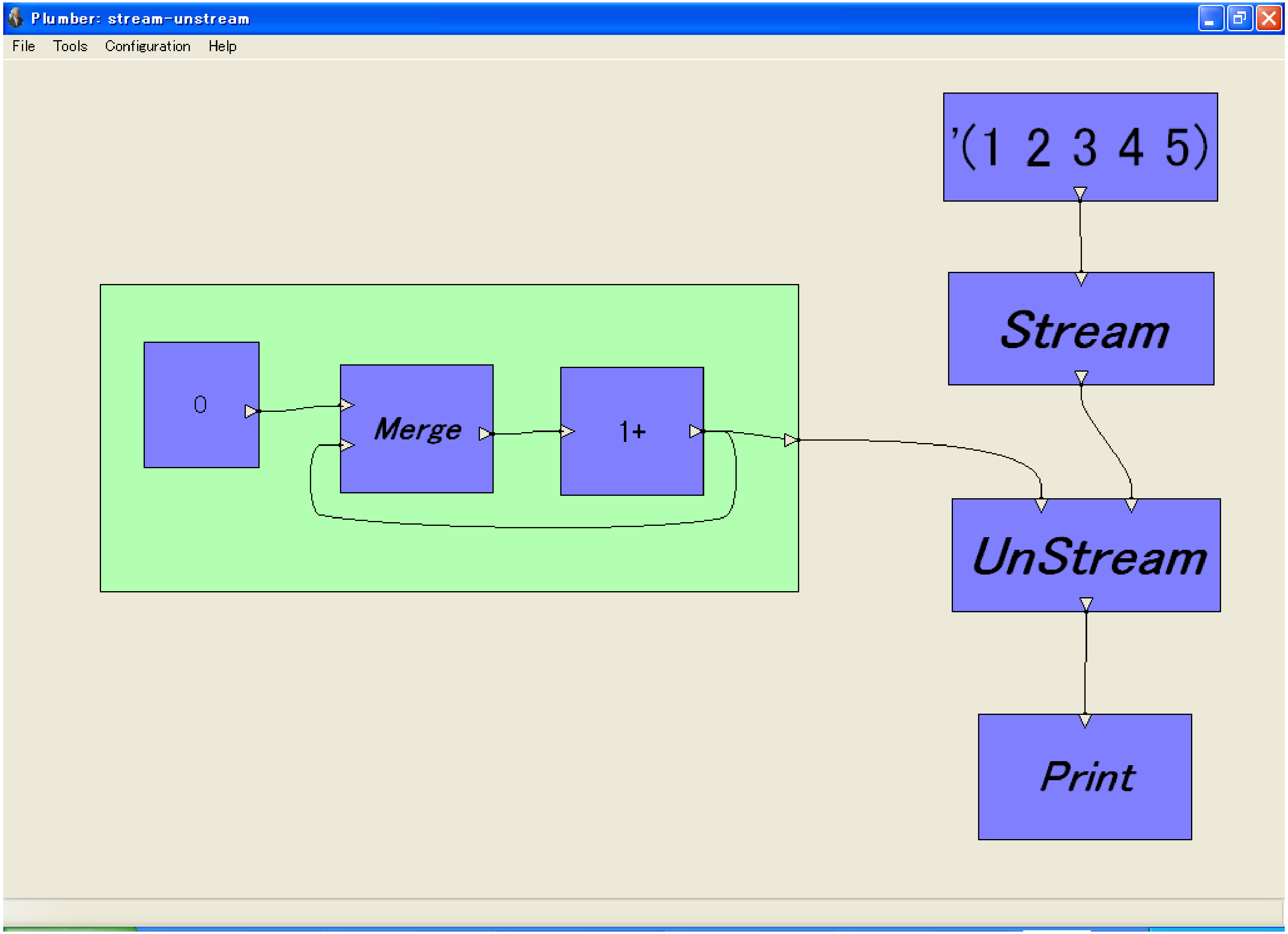
3/5. Streams

Stream operations in Plumber

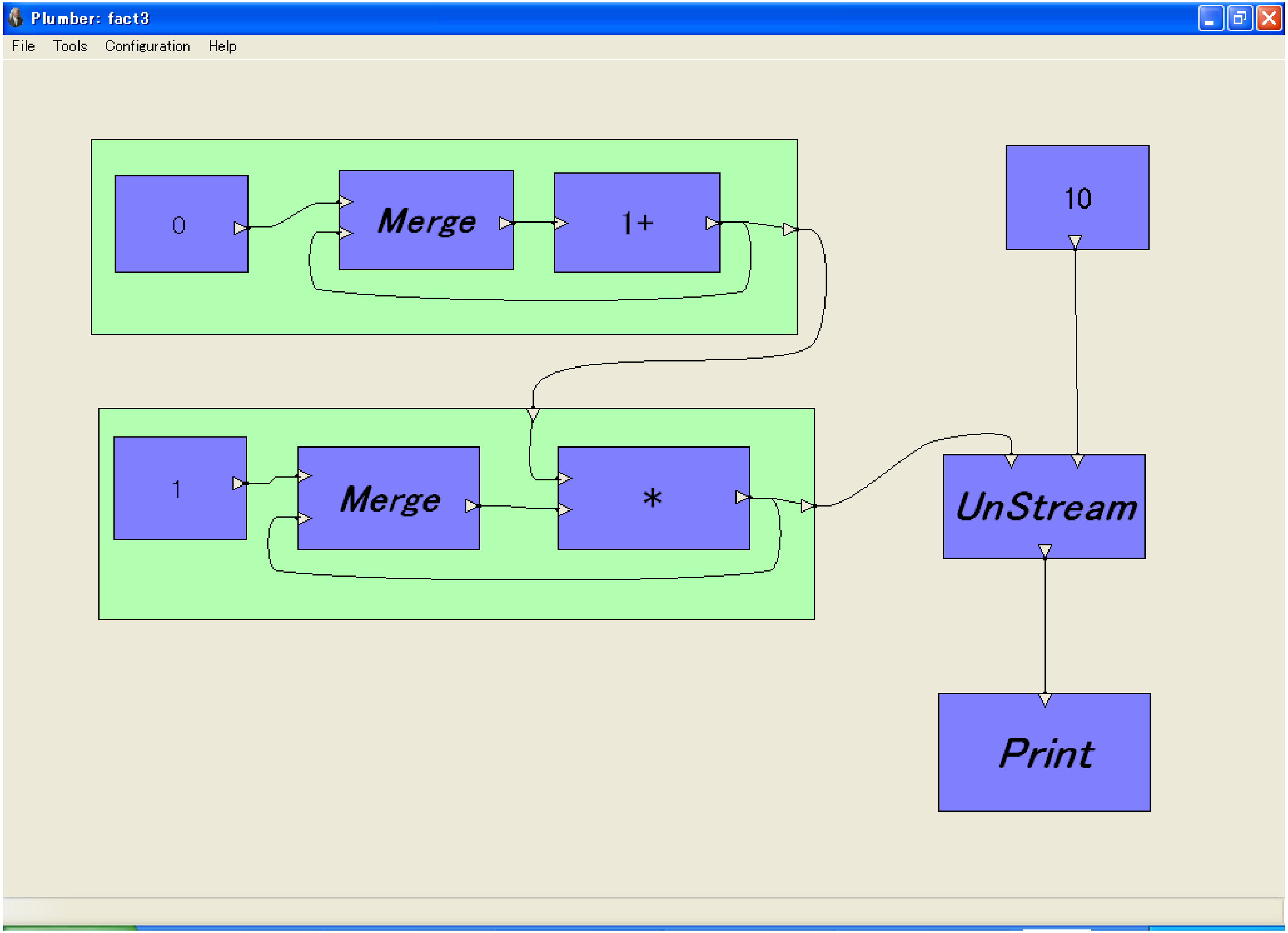
- Stream
  - Converts a list to a stream
- UnStream
  - Converts a stream to list
- Repeat
  - Make a stream by repeating a data

Demonstration 2

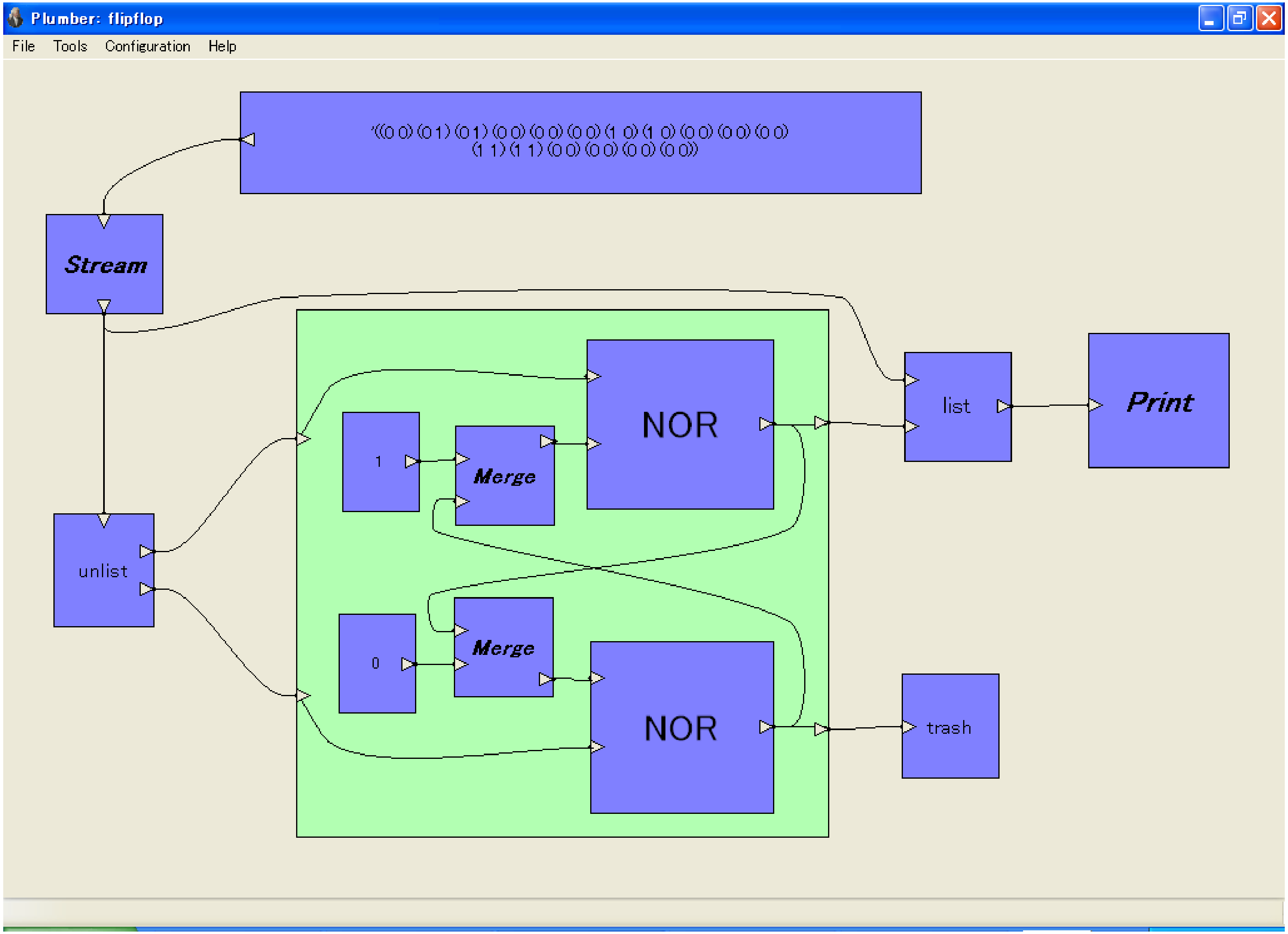
stream-unstream



fact3

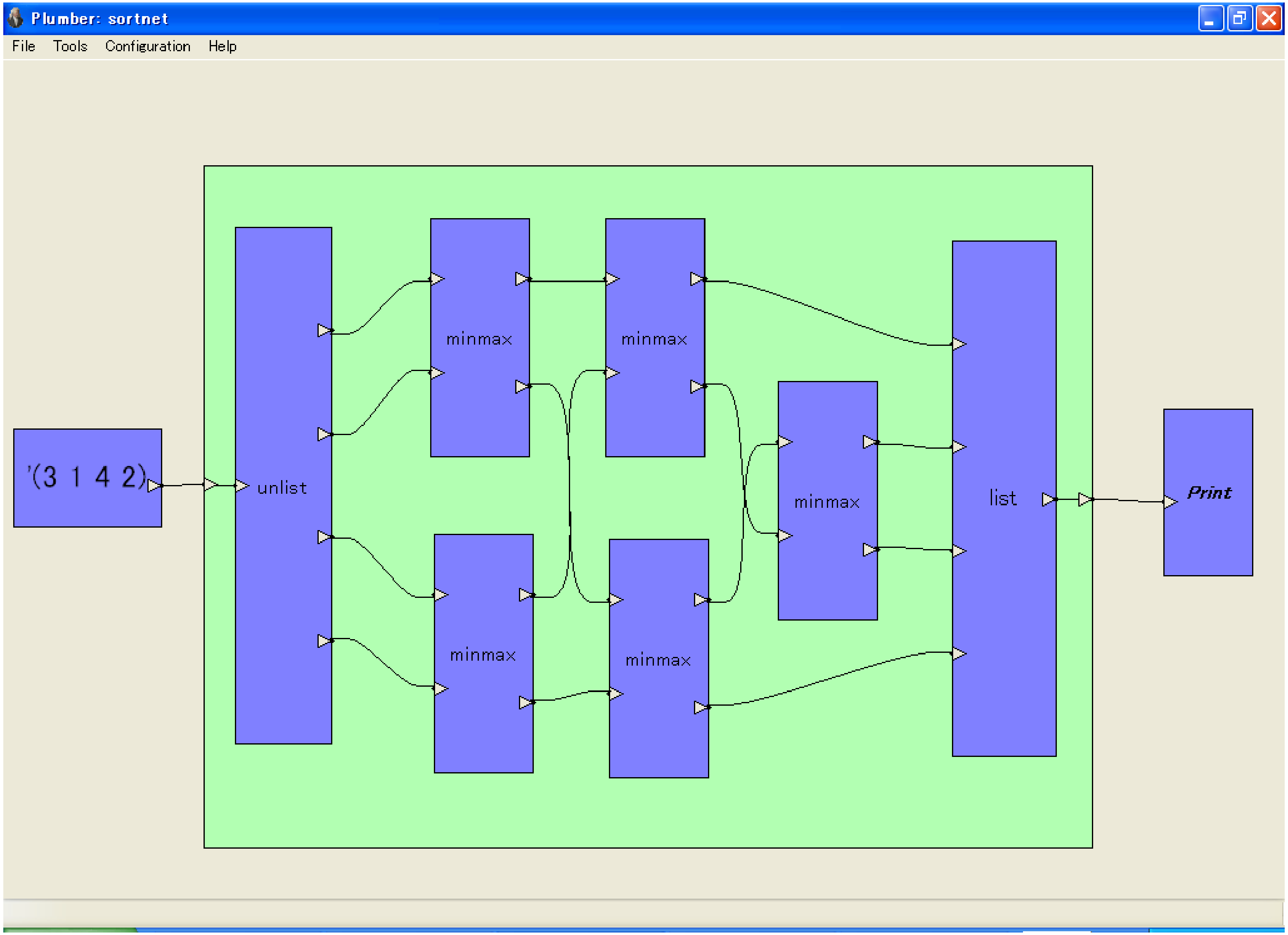

Plumber: fact3
File
Tools
Configuration
Help
0
Merge
1+
10
1
Merge
*
UnStream
Print

flipflop



```
'((0 0) (0 1) (0 1) (0 0) (0 0) (0 0) (1 0) (1 0) (0 0) (0 0) (0 0)
(1 1) (1 1) (0 0) (0 0) (0 0) (0 0))
```

sortnet

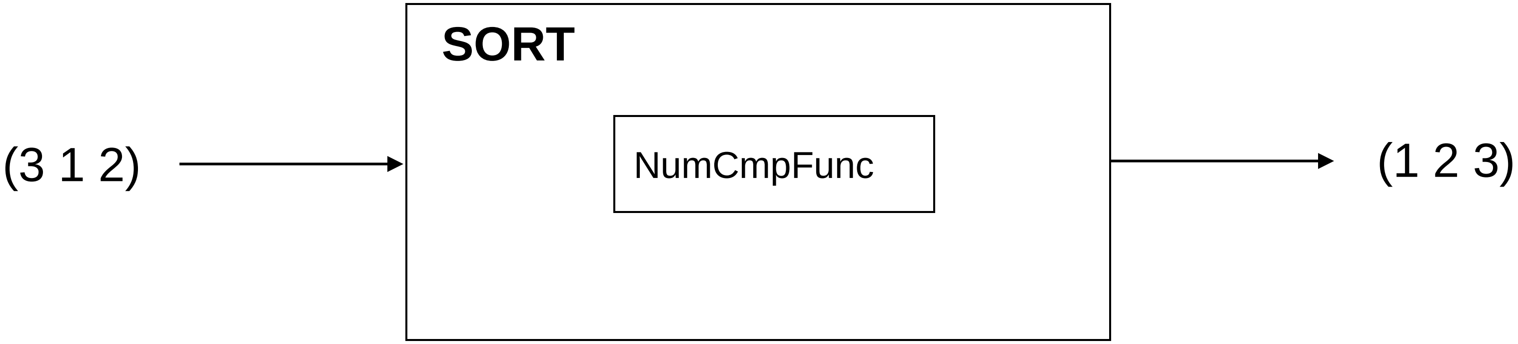

Plumber: sortnet
File
Tools
Configuration
Help
'(3 1 4 2)
unlist
minmax
minmax
minmax
minmax
minmax
list
Print

4/5. Higher order functions

Previous approaches for higher order functions in DFVPL

- Special node with function hole approach
  - Easy to understand for non-programmers
  - Seems awkward for Lispers
  - Used in: DataVis, CUBE, ESTL, YUBA
- Function as a value approach
  - Seems difficult for non-programmers
  - Needs a node for funcall or apply
  - Used in: ShowAndTell, VPL, PHF
  - Plummer also impremented this automatically

Special node with function hole approach



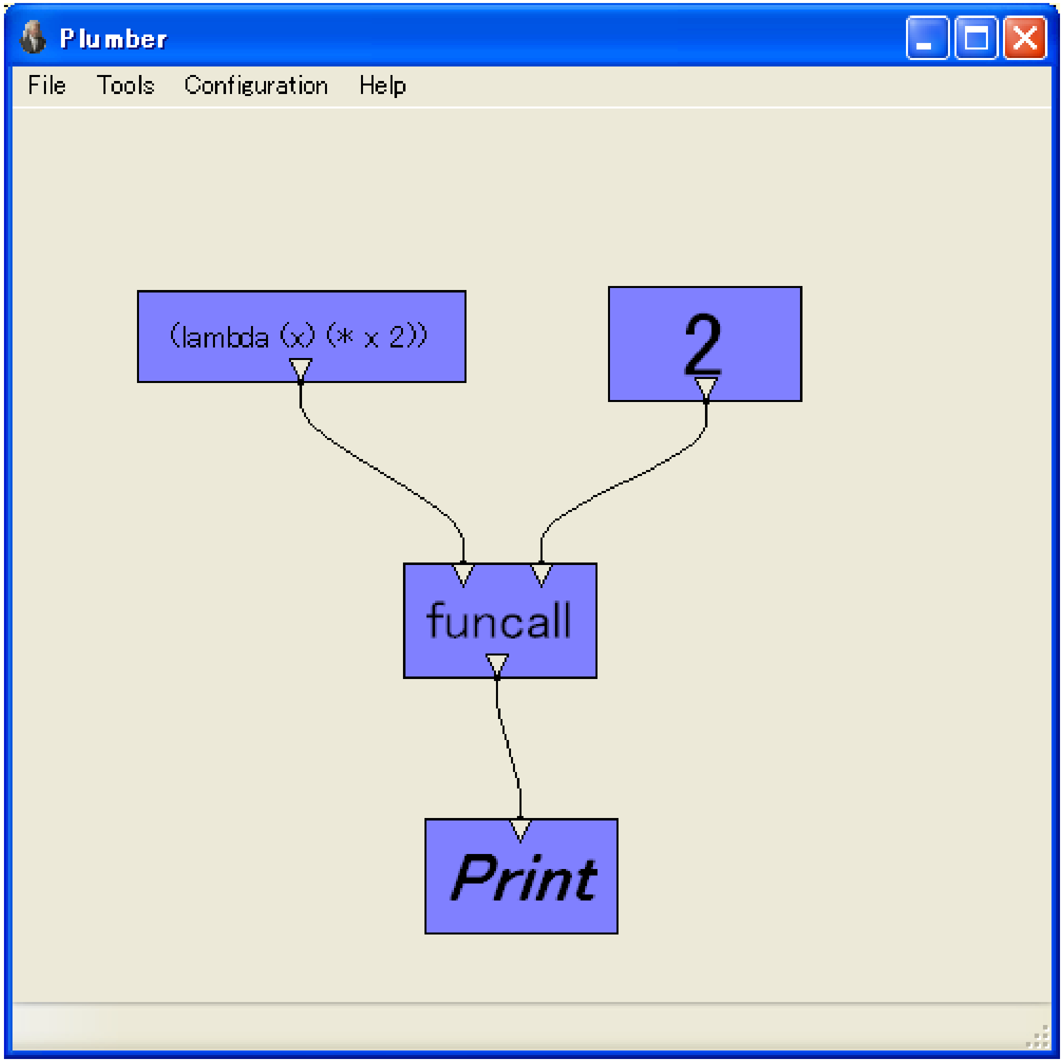
(3 1 2)

SORT

NumCmpFunc

(1 2 3)

Function as a value approach



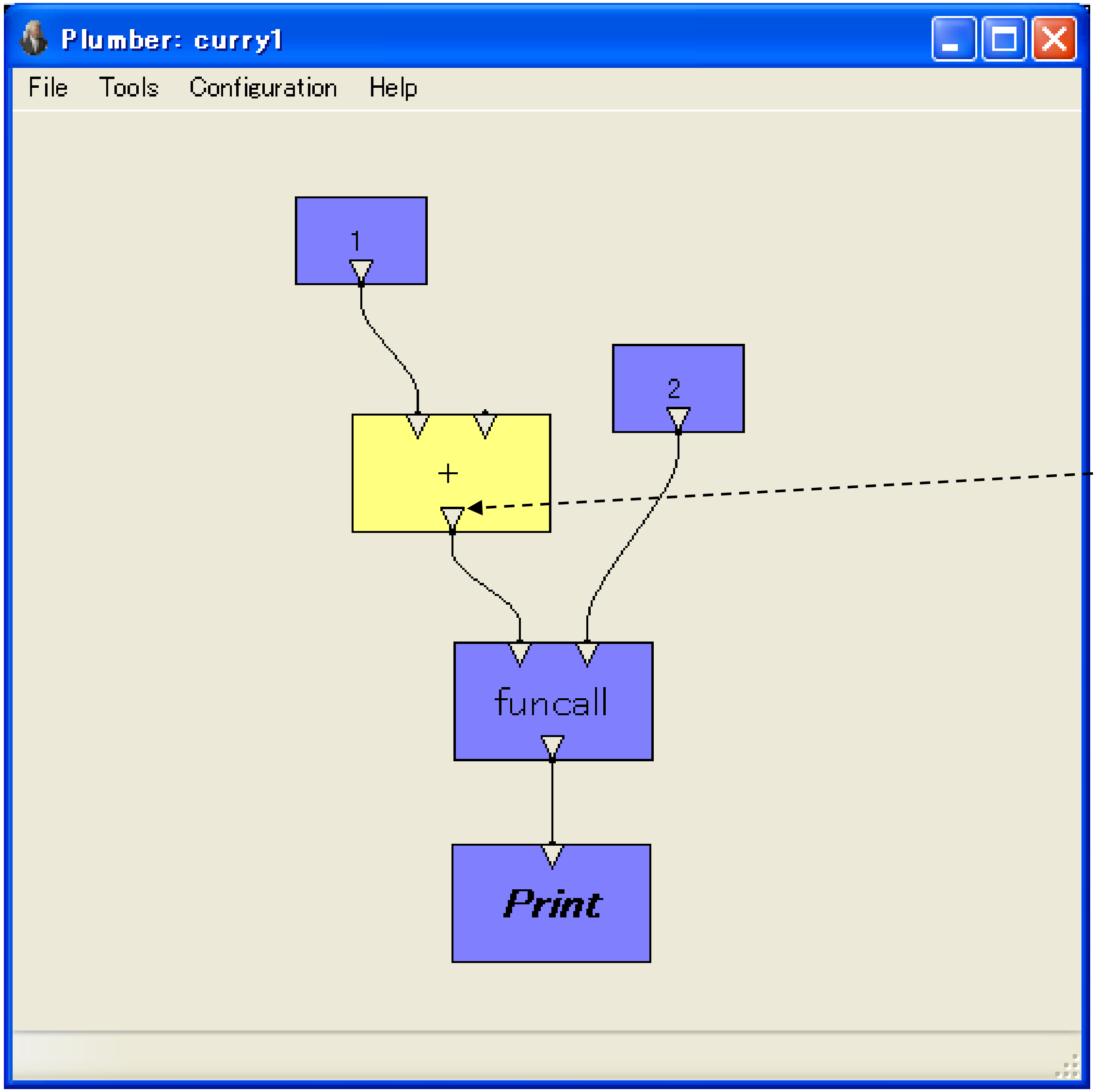
Drawbacks of the simple version of function as a value approach

1. It depends the power of Lisp too much
2. Making a function value from a previously defined function is impossible

What is the most suitable notation for DFVPL ?

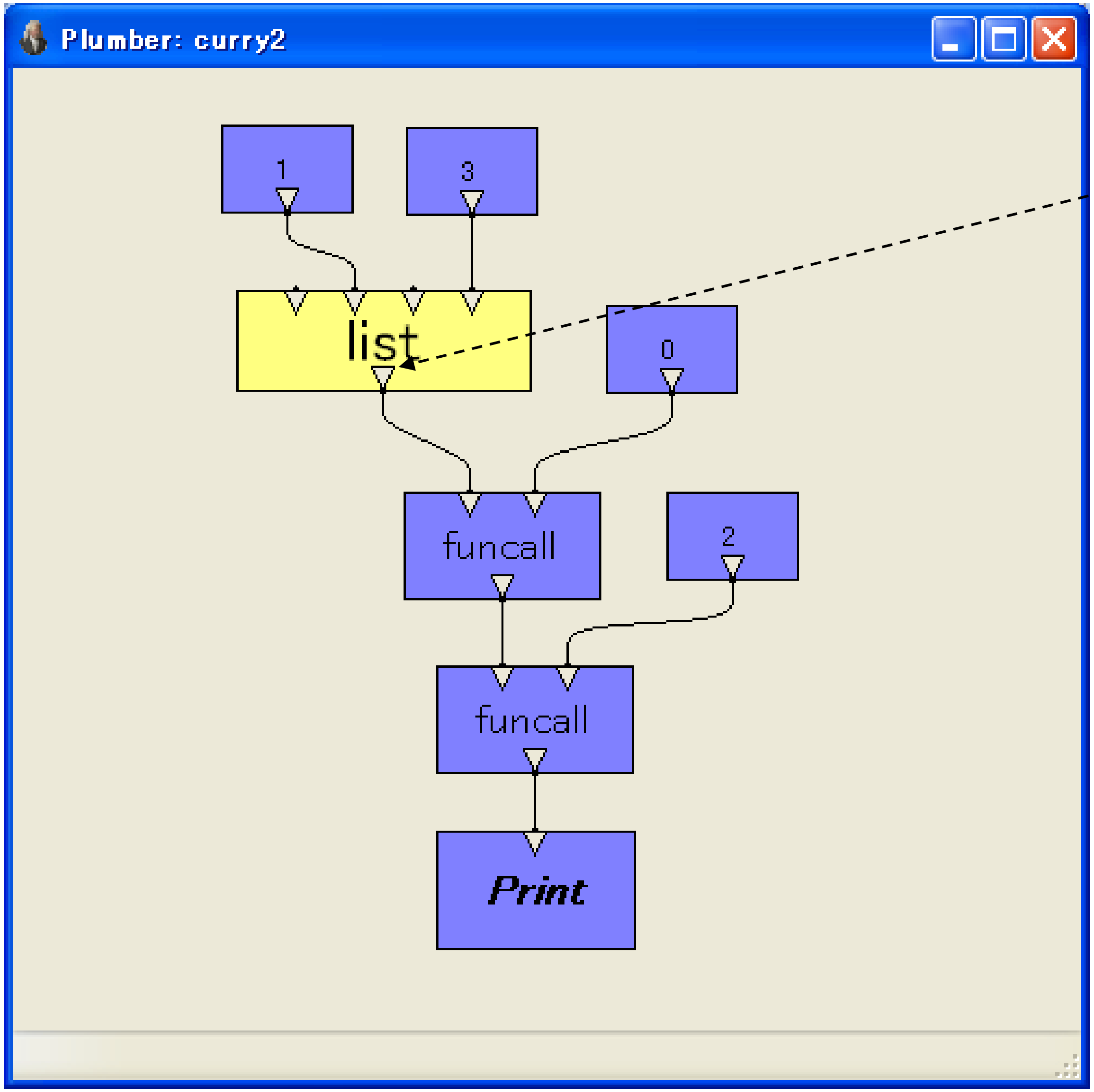
Our answer is the currying based notation !

curry1



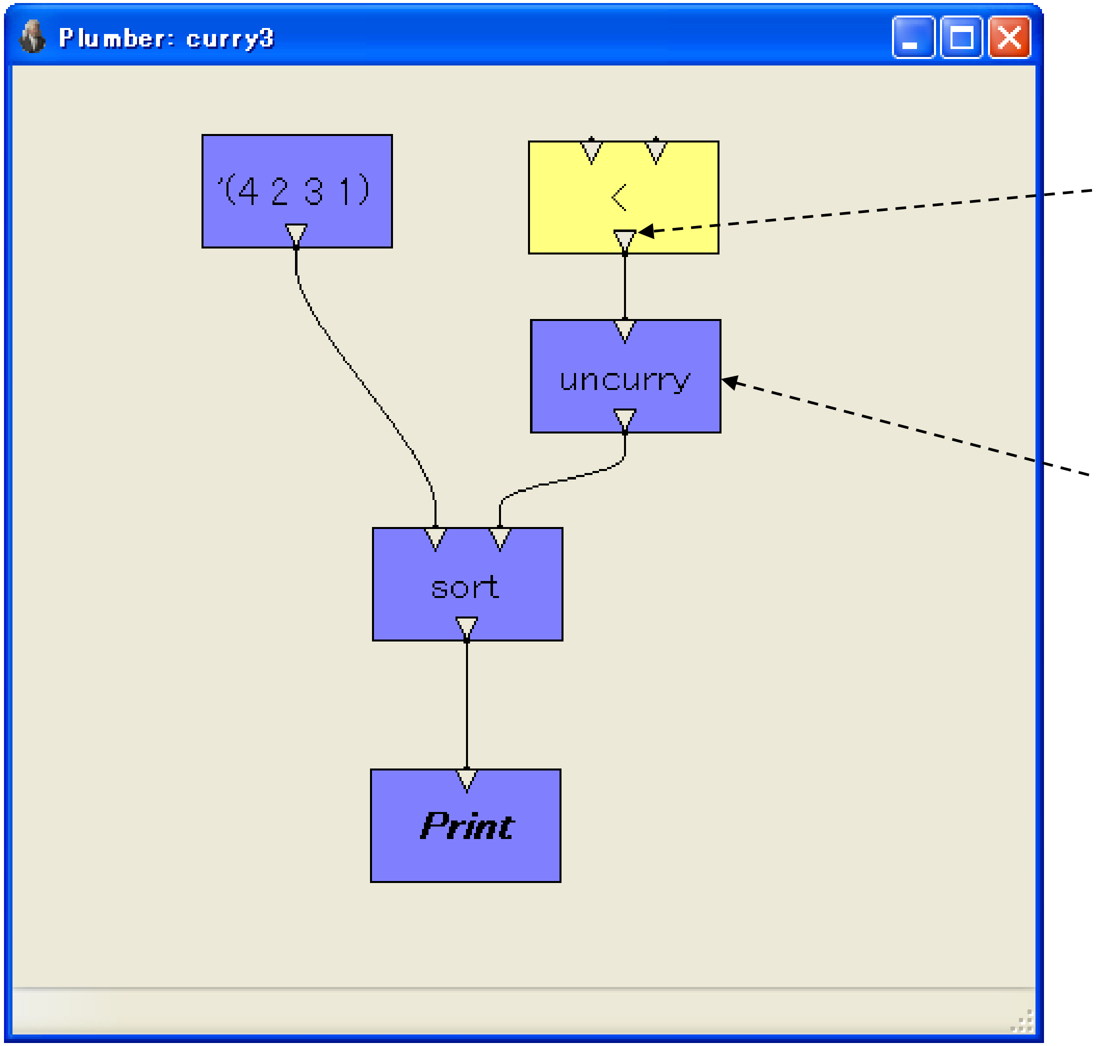
```
(lambda (x0)
  (lambda (x1)
    (+ x0 x1)))
```

curry2



```
(lambda (x1 x3)
  (lambda (x0)
    (lambda (x2)
      (list x0 x1 x2 x3))))
```

curry3



```
(lambda ()
  (lambda (x0)
    (lambda (x1)
      (< x0 x1))))
```

```
(defun uncurry (fun)
  (lambda (x y)
    (funcall
      (funcall fun x)
     y)))
```

5/5. Finding monads

Monads in the literature

- E.Moggi, **Computational lambda-calculus and monads**, 1989.

- P.Wadler, **The essence of functional programming**, 1991.

- J.Hughes, **Generalizing monads to arrow**, 2000.

Value with effect $\mathrm{M}\alpha$

$$\mathrm{M}\alpha = (\alpha \quad \text{ListOfDebugDumpString})$$

```
(defun Print! (Ma)
  ;; prints purely part value
  (format t "value = ~s~%~%" (car Ma))
  ;; prints debug dump output
  (dolist (str (cadr Ma))
    (format t "~a~%" str)))
```

(sqrt! (1+! 1)) ?

$$1+! : \text{Integer} \rightarrow \text{M Integer}$$

returns argument+1   with debug dump

```
(defun 1+! (x)
  (let ((y (1+ x)))
    (list y (list (format nil "~s ->1+!-> ~s" x y)))))
```

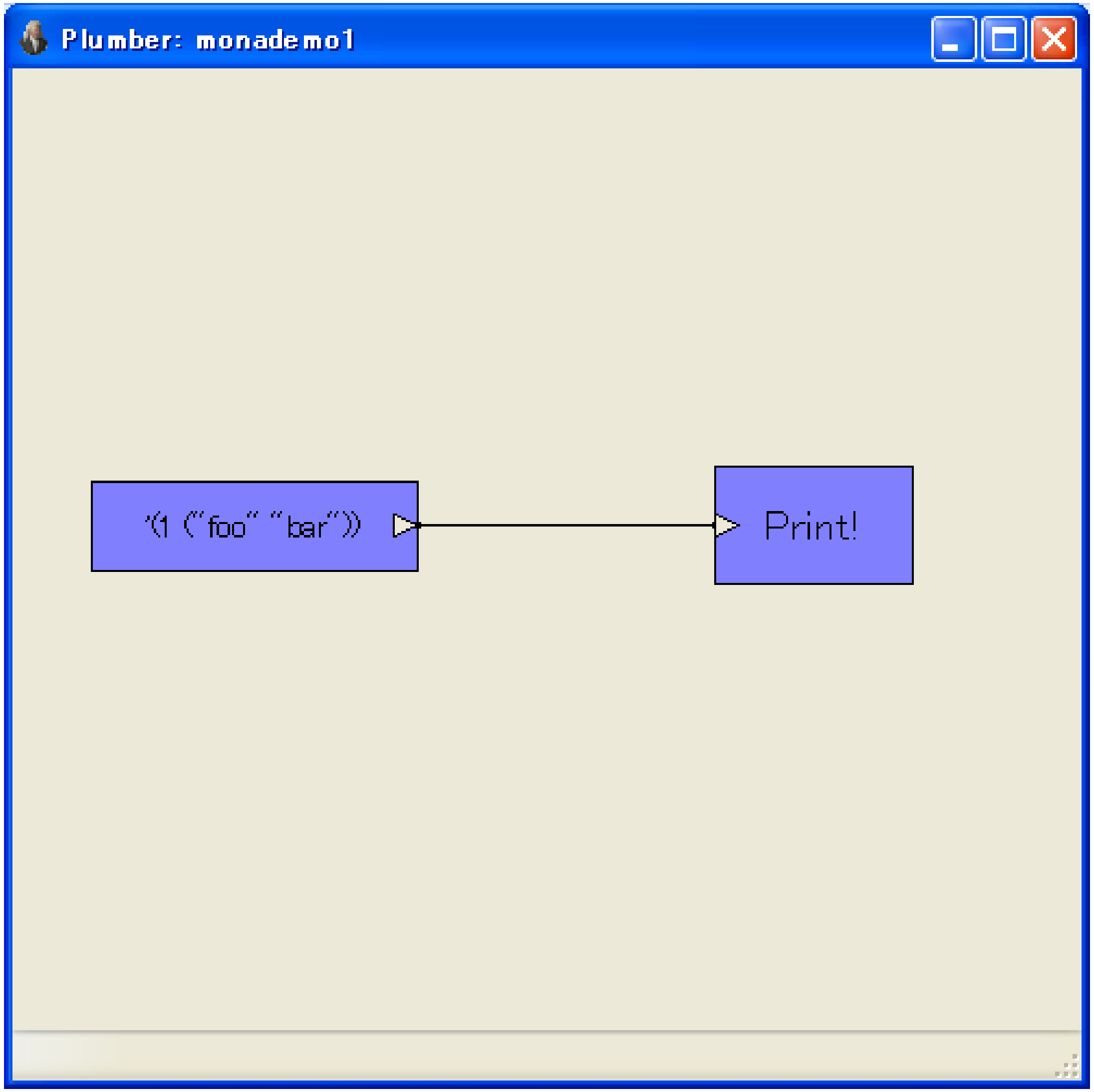
$$\text{sqrt!} : \text{Integer} \rightarrow \text{M Real}$$

returns sqrt of argument with debug dump

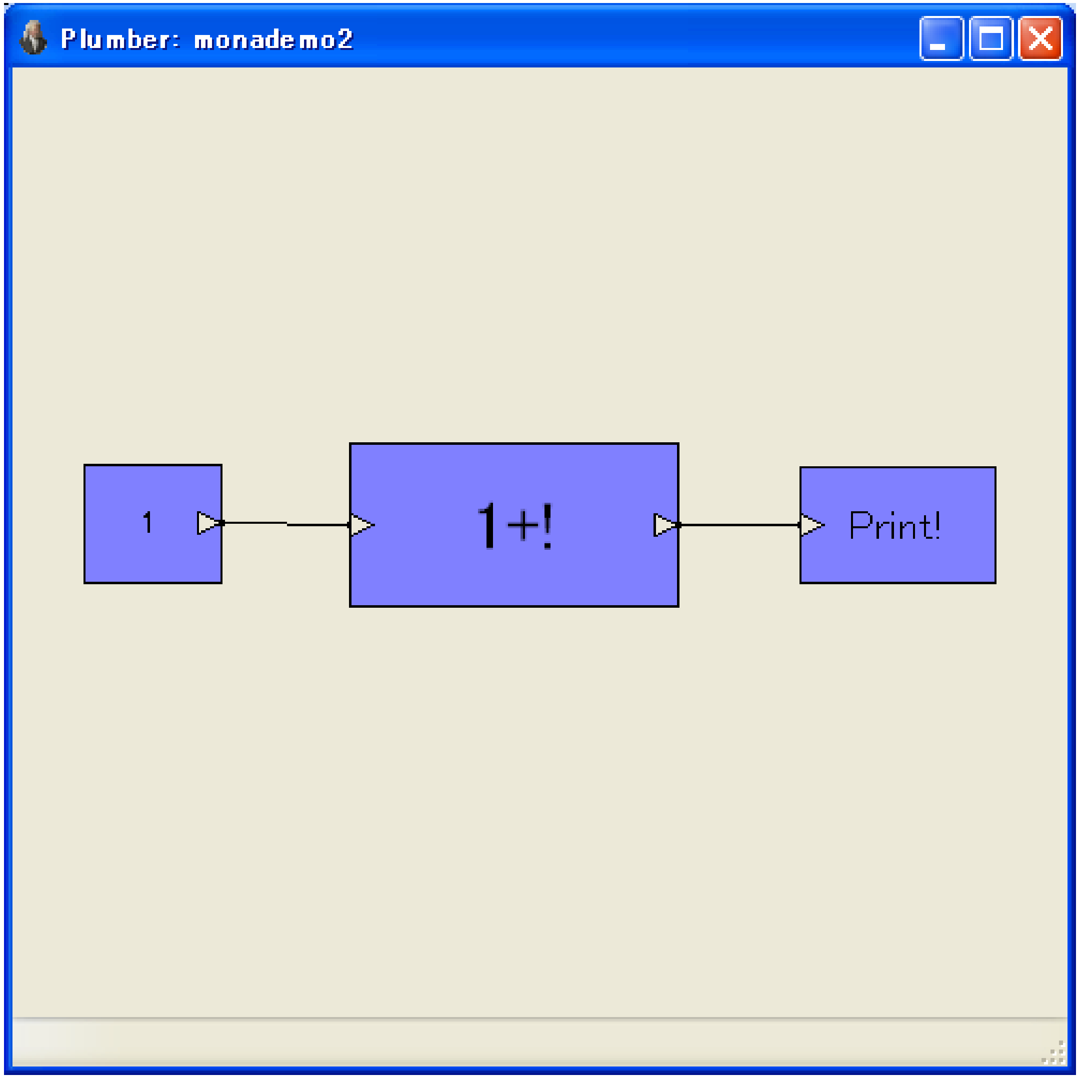
```
(defun sqrt! (x)
  (let ((y (sqrt x)))
    (list y (list (format nil "~s ->sqrt!-> ~s" x y)))))
```

Demonstration 3

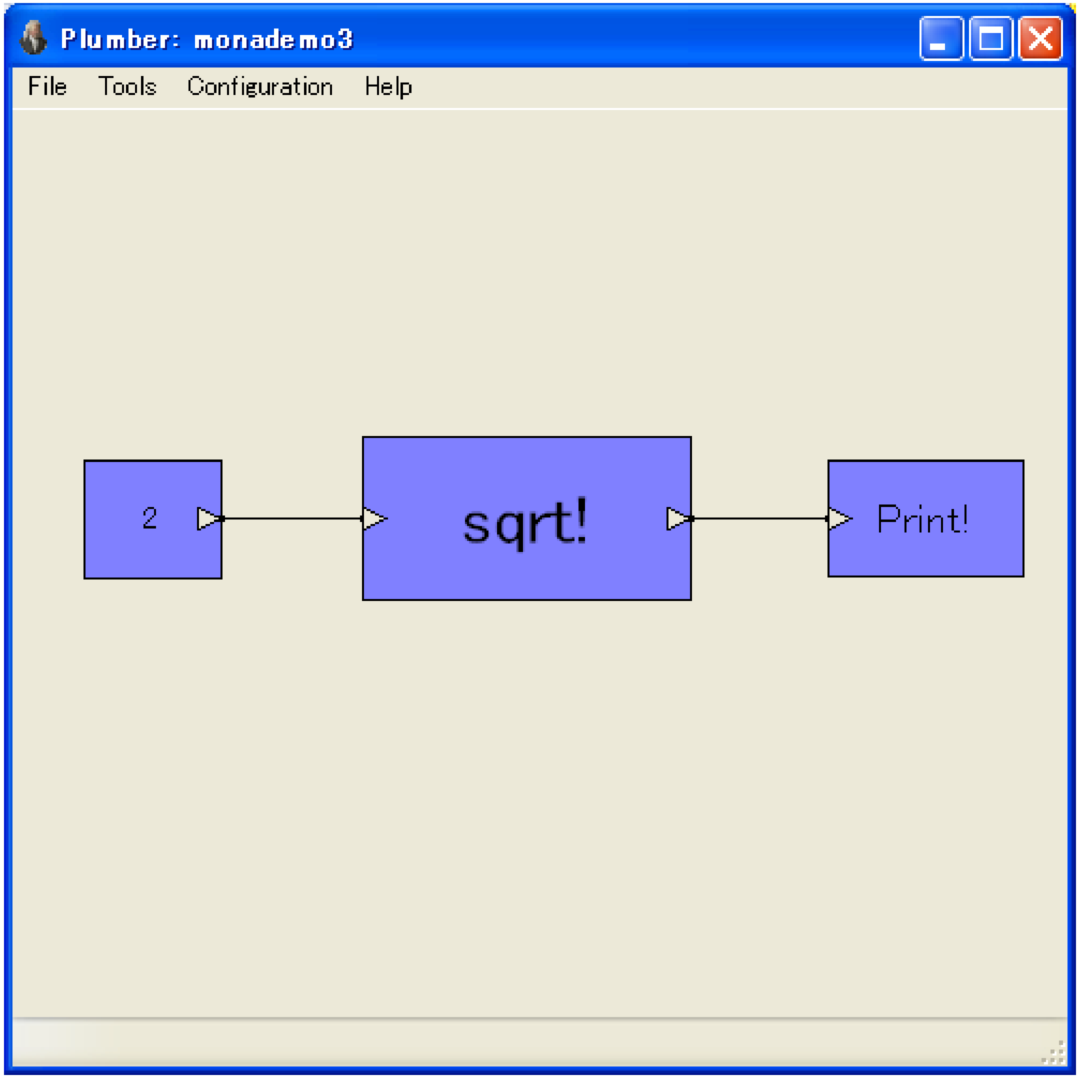
monademo1



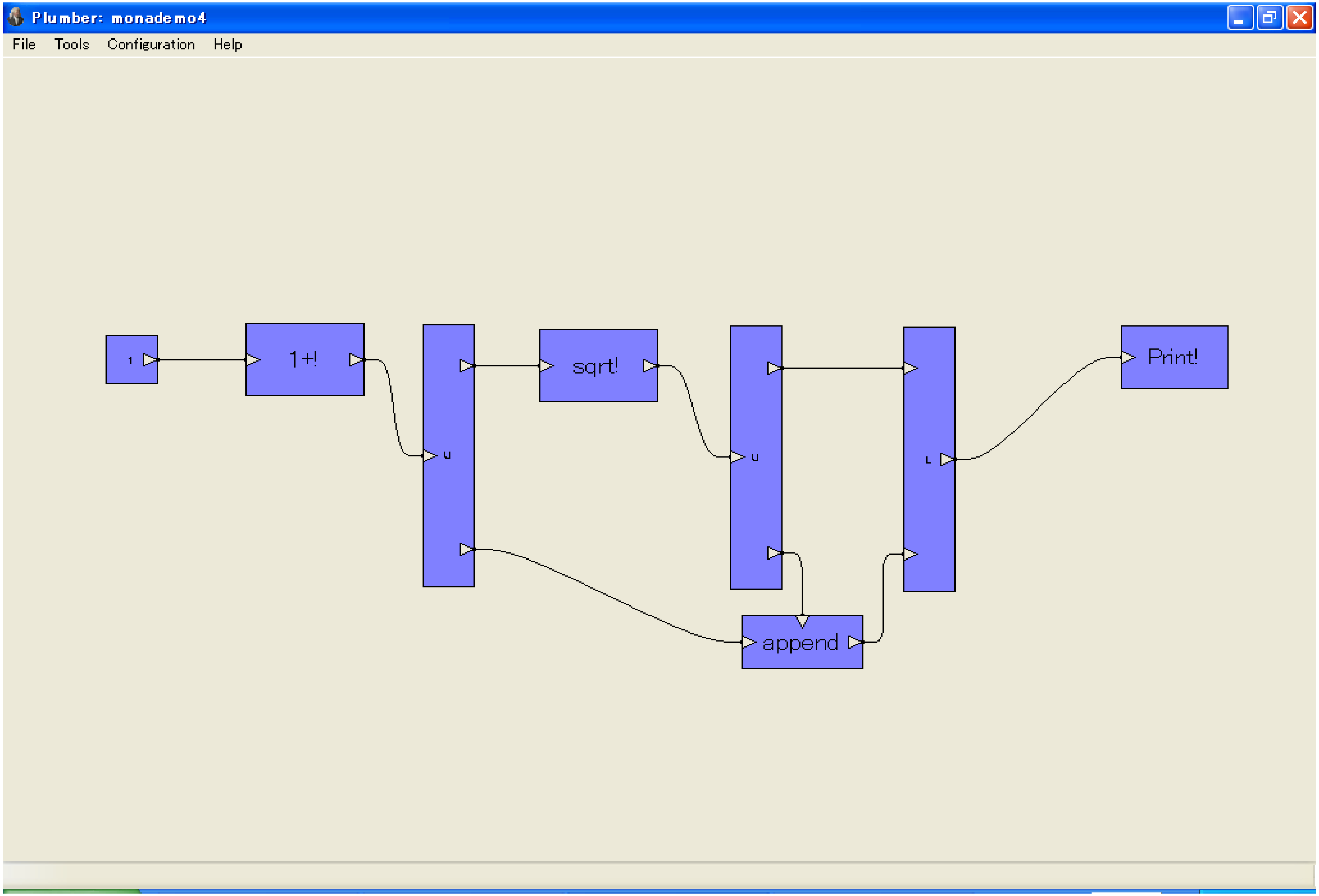
monademo2



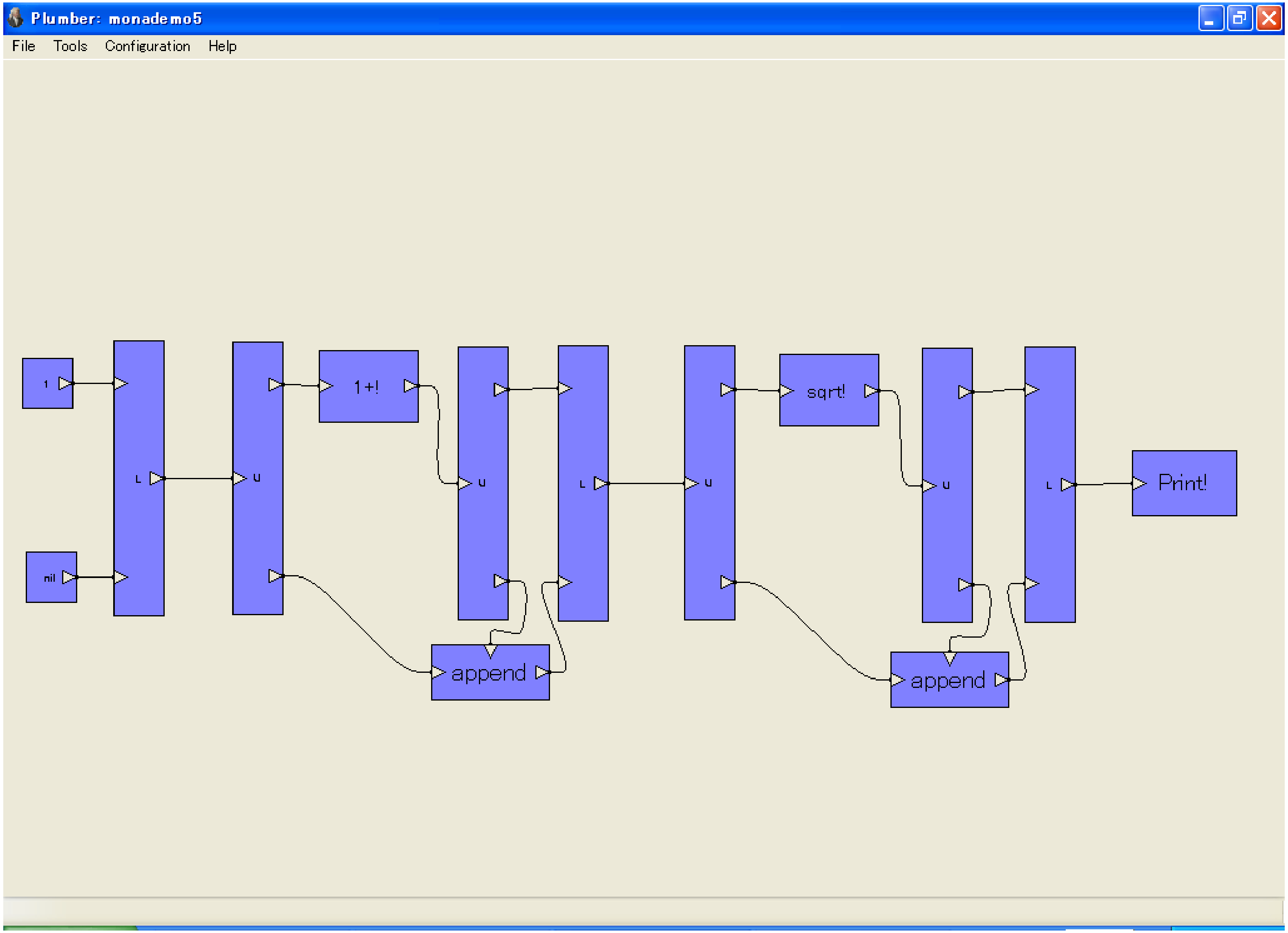
monademo3


Plumber: monademo3
File
Tools
Configuration
Help
2
sqrt!
Print!

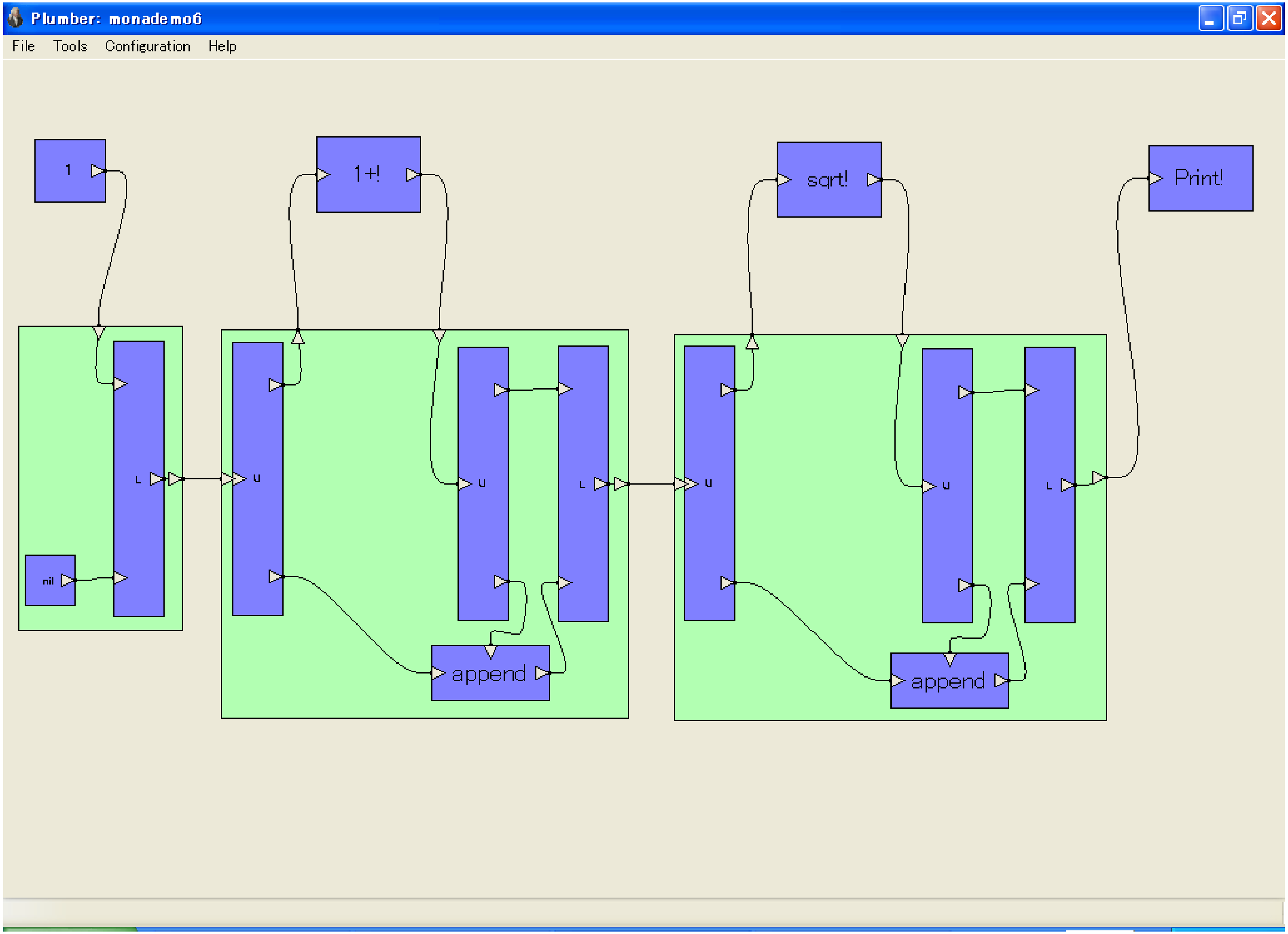
monademo4


Plumber: monademo4
File
Tools
Configuration
Help
1
1+!
u
sqrt!
u
L
Print!
append

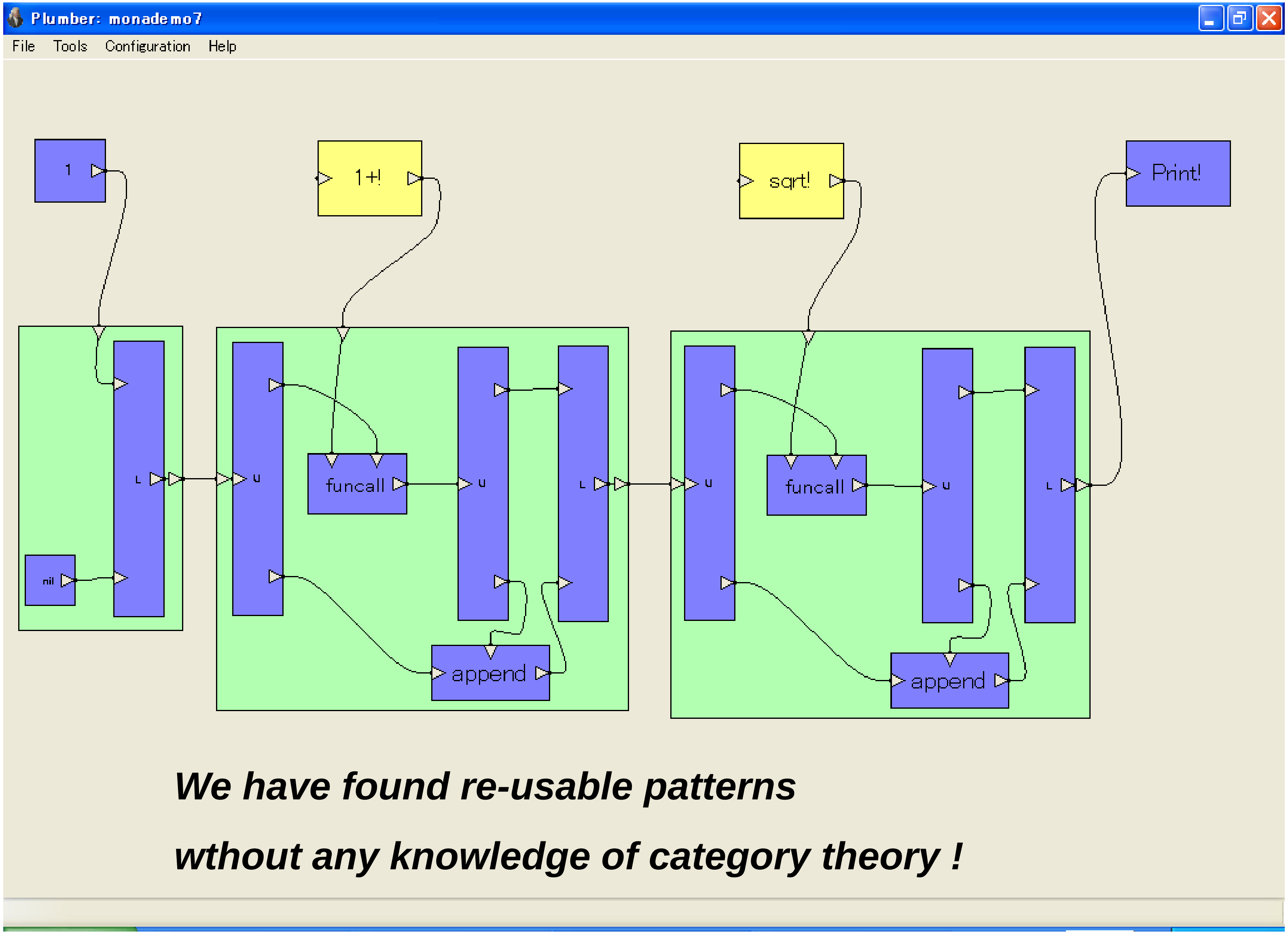
monademo5


Plumber: monademo5
File
Tools
Configuration
Help
1
nil
L
u
1+!
u
L
u
sqrt!
u
L
Print!
append
append

monademo6


Plumber: monademo6
File
Tools
Configuration
Help
1
1+!
sqrt!
Print!
nil
L
U
U
L
U
U
L
append
append

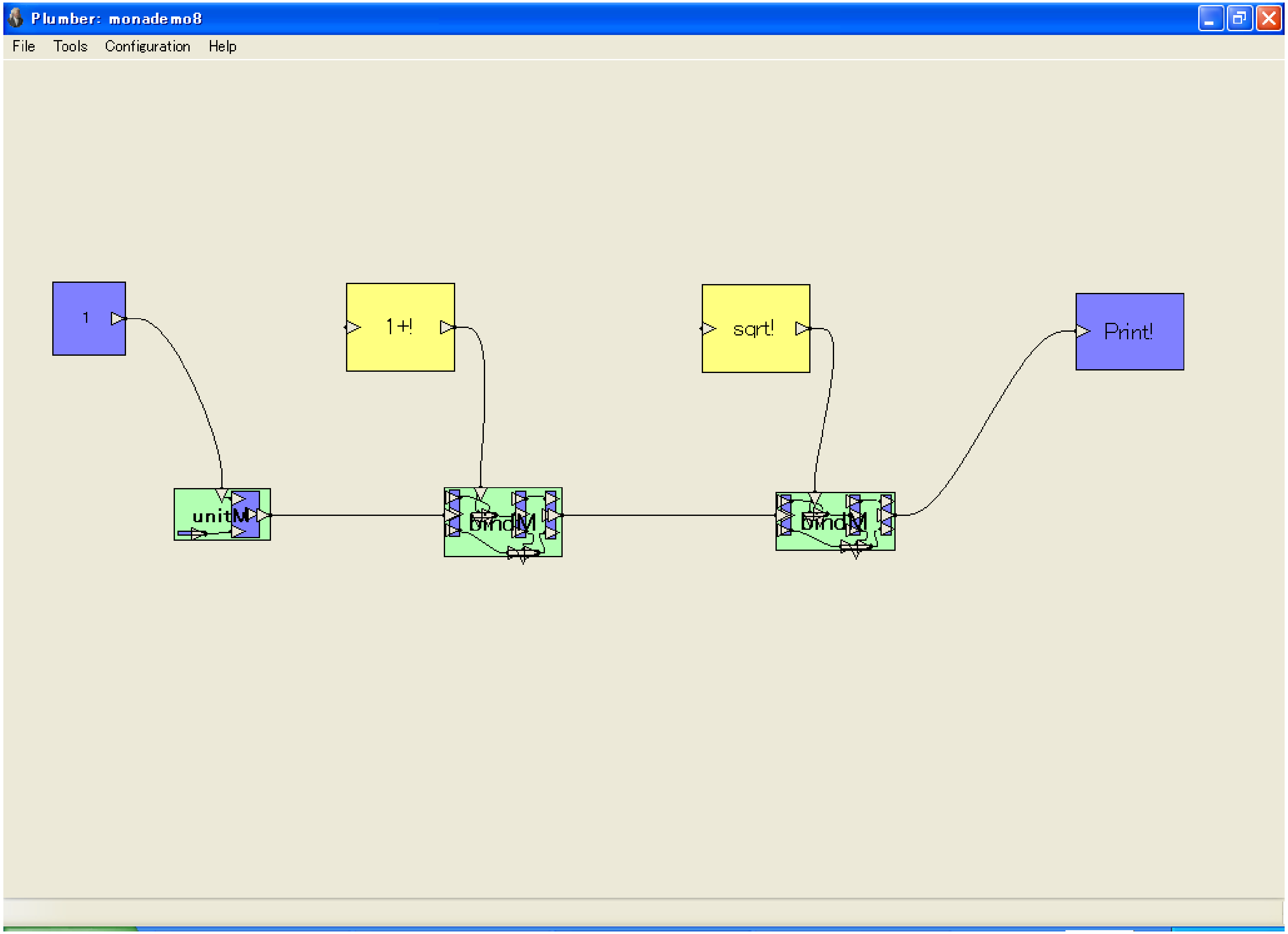
monademo7



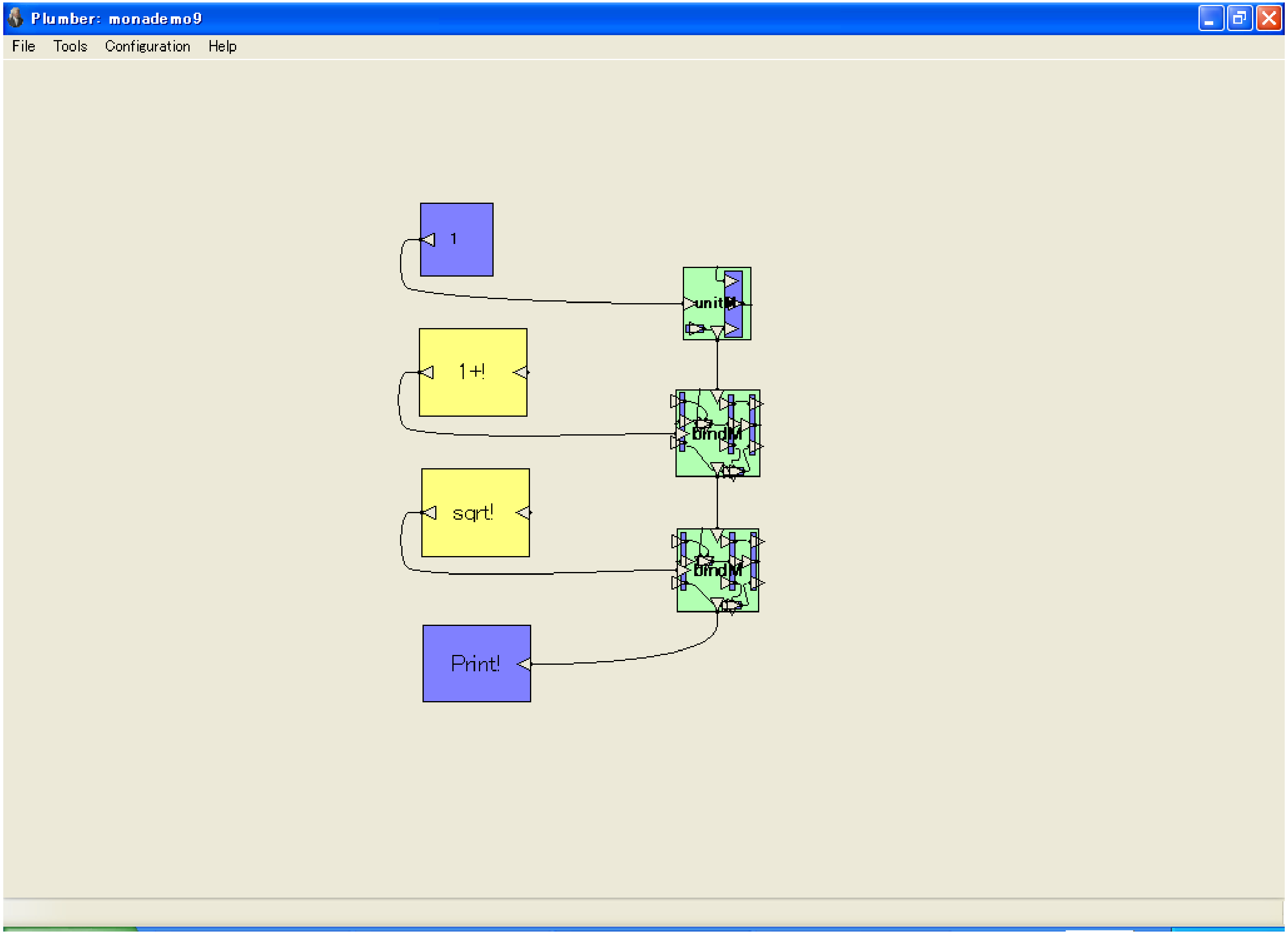
We have found re-usable patterns

wthout any knowledge of category theory !

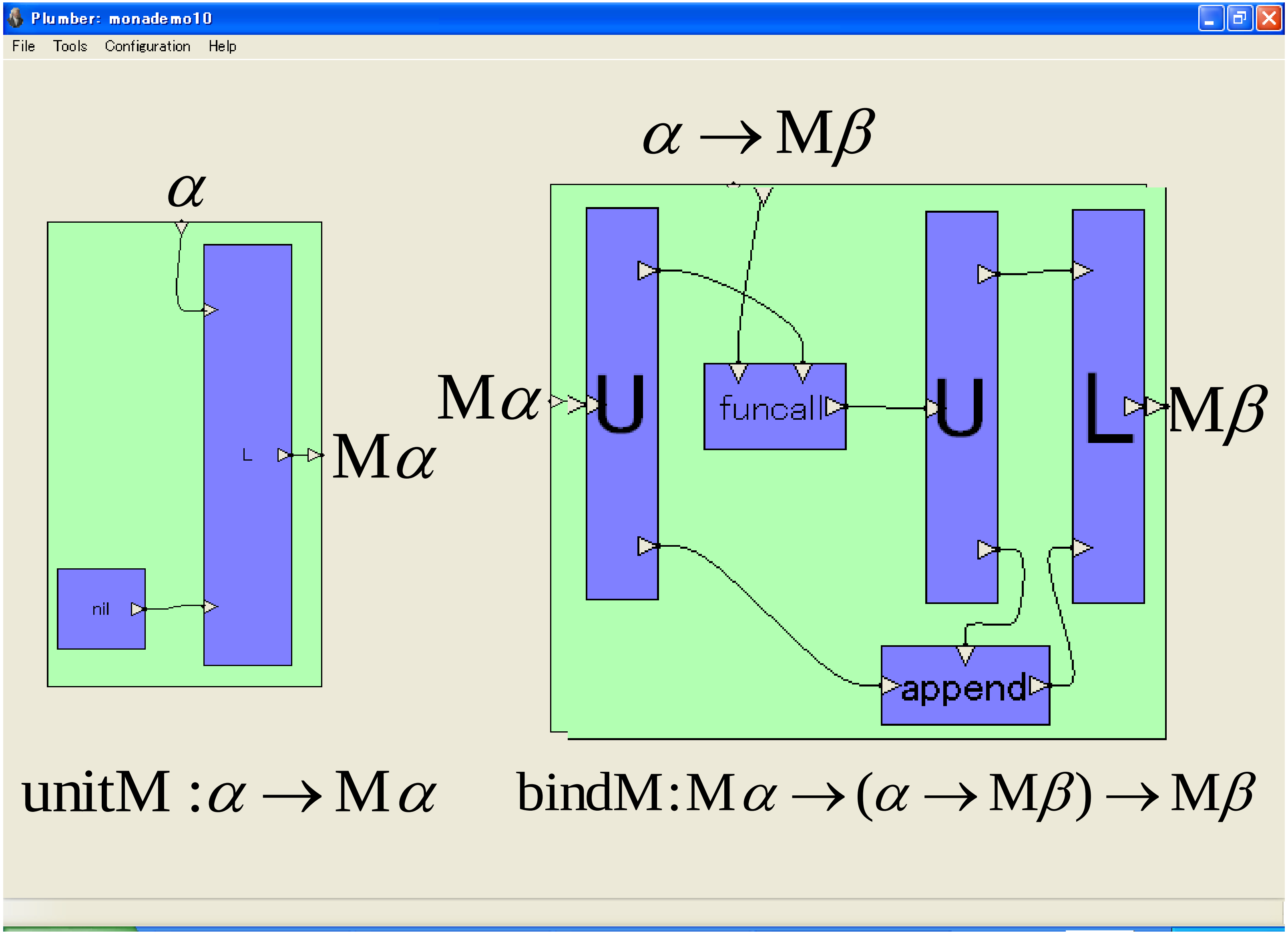
monademo8


Plumber: monademo8
File
Tools
Configuration
Help
1
1+!
sqrt!
Print!
unitM
bindM
bindM

monademo9


Plumber: monademo9
File
Tools
Configuration
Help
1
unitM
1+!
bindM
sqrt!
bindM
Print!

Correspondence to the definition in the literature



$$\mathrm{unitM} : \alpha \rightarrow M\alpha$$

$$\mathrm{bindM} : M\alpha \rightarrow (\alpha \rightarrow M\beta) \rightarrow M\beta$$

Final remarks

- Plumber is firstly designed for non-programmers
- And intended to provide a customize features for MSI products, as in Emacs Lisp of Emacs
- It is still incomplete as a general-purpose programming language
- This project is being suspended over two years

Thank you !

Monad laws

$$\mathrm{unitM} : \alpha \to \mathrm{M}\alpha$$

$$\mathrm{bindM} : \mathrm{M}\alpha \to (\alpha \to \mathrm{M}\beta) \to \mathrm{M}\beta$$

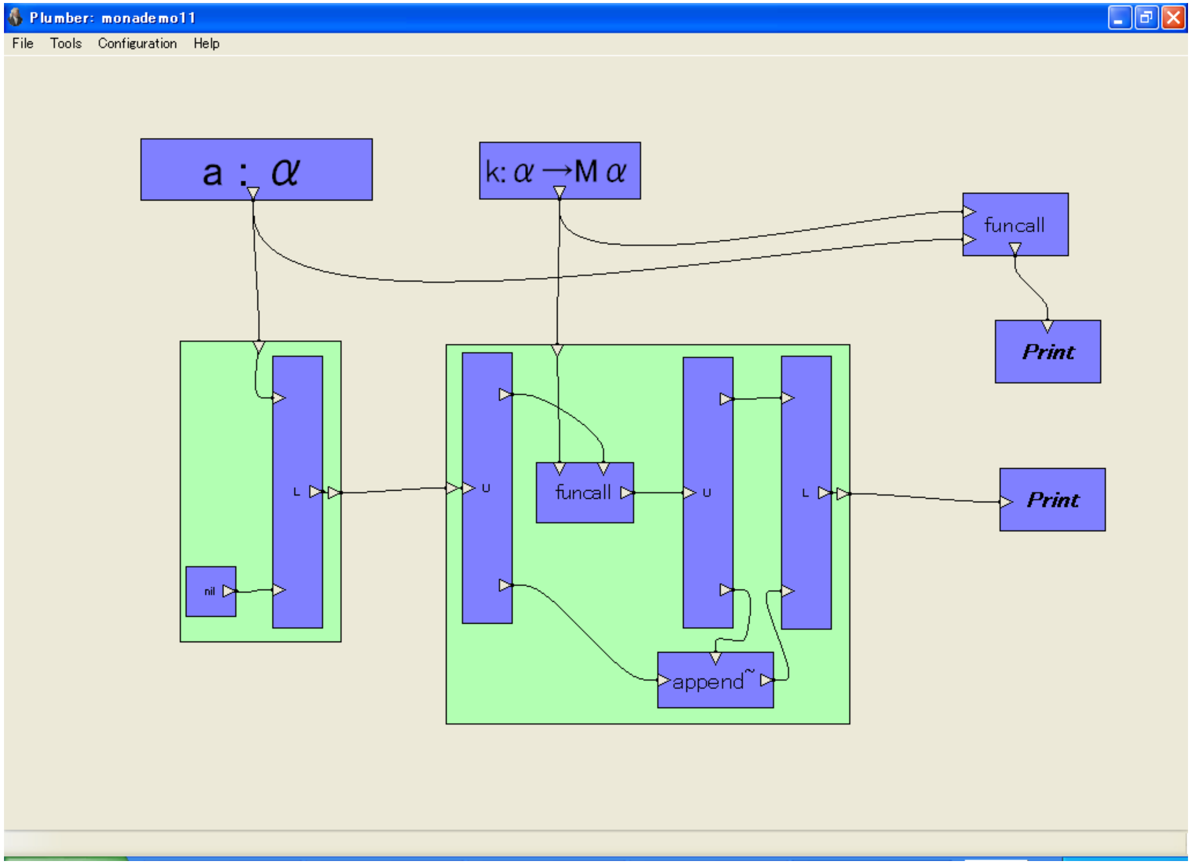
$$(\mathrm{unitM}\; a)\ \text{`bindM`}\ k = k a$$

$$m\ \text{`bindM`}\ \mathrm{unitM} = m$$

$$m\ \text{`bindM`}\ (\lambda a.(k a\ \text{`bindM`}\ h))$$

$$= (m\ \text{`bindM`}\ \lambda a.k a)\ \ \text{`bindM`}\ h$$

$(\text{unitM}\ a)\ \text{`bindM`}\ k = ka$



```
value = ((f a) ("string"))
value = ((f a) (append nil ("string")))
```

`(append nil x) = x`

$$m \text{ `bindM` unitM} = m$$

⇓

`(append x nil) = x`

$$
\begin{aligned}
& m\ \text{`bindM`}\ (\lambda a.(ka\ \text{`bindM`}\ h)) \\
= & (m\ \text{`bindM`}\ \lambda a.ka)\ \ \text{`bindM`}\ h
\end{aligned}
$$

⇓

```
  (append x (append y z))
= (append (append x y) z))
```

Monad laws of $\mathrm{M}\alpha$

⇓

(リスト，`nil`，`append`)

はモノイドでなければならない！