

Playing Landin's Game

Better Programs by Reducing Explicit Sequencing



Eric Smith, LambdaJam 2014

@eric_s_smith

github: [ericssmith/presentations/blob/master/landinsgame.pdf](https://github.com/ericssmith/presentations/blob/master/landinsgame.pdf)

“Now why is [go to] so ugly? Because it is inelegant and uneconomical. A go to statement is about the most empty construction conceivable. Executing it is like being told: It is not me, it is my colleague!” [1]

- *Peter Naur*
BIT (1963)

Separation Anxiety

- goto
- if
- while
- for
- null
- globals
- tests
- runtime verification

“Now why is [go to] so ugly? Because it is inelegant and uneconomical. A go to statement is about the most empty construction conceivable. Executing it is like being told: It is not me, it is my colleague! ...

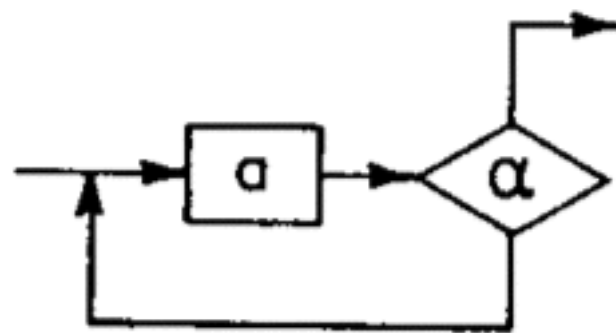
Well, if you look carefully you will find that surprisingly often **a go to statement which looks back really is a concealed *for* statement**. And you will be pleased to find how the clarity of the algorithm improves when you insert the *for* clause where it belongs.” [1]

- *Peter Naur*
BIT (1963)

“It is thus proved possible to completely describe a program by means of a formula containing the names of diagrams Φ , Π , and Δ , ” [2, but see 3]

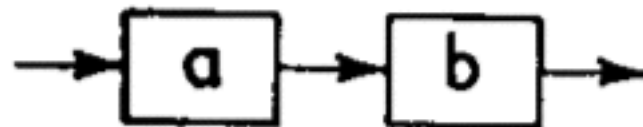
- *Giuseppe Jacopini*

Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules
(1966)



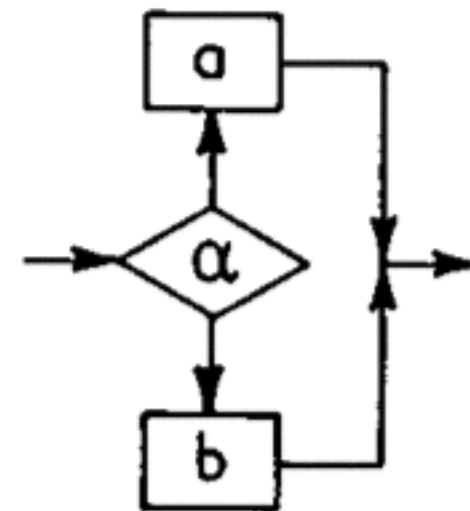
Φ

iteration



Π

composition



Δ

alternation

“10. Eliminating Explicit Sequencing

There is a game sometimes played with ALGOL 60 programs--rewriting them so as to avoid using labels and go to statements. It is part of a more embracing game -- **reducing the extent to which the program conveys its information by explicit sequencing**. Roughly speaking this amounts to using fewer and larger statements. The game's significance lies in that it frequently produces a more "transparent" program--easier to understand, debug, modify and incorporate into a larger program.

The author does not argue the case against explicit sequencing here. Instead he takes as point of departure the observation that **the user of any programming language is frequently presented with a choice between using explicit sequencing or some alternative feature of the language.**” [4]

- *Peter Landin*

The Next 700 Programming Languages (1966)

“Programming languages ... came under pressure to allow ordinary mathematical expressions as well as the elementary commands. It is, after all, much more convenient to write ... $x := a(b+c)+d$ than the more elementary

CLA b
ADD c
MPY a
ADD d
STO x

implicit sequencing



explicit sequencing



” [5]

- Christopher Strachey
Fundamental Concepts in Programming Languages (1967)

“Von Neumann programming languages use variables to imitate the computer's storage cells; **control statements elaborate its jump and test instructions**; and assignment statements imitate its fetching, storing, and arithmetic.” [6]

- *John Backus*

Can Programming Be Liberated from the von Neumann Style? (1978)

- Studied Lambda Calculus at Cambridge.
- 1954 - First exposure to computers was witnessing the EDSAC “stumbling through Runge-Kutta”. Decided “this was not for me”.
- But later he got job doing statistical analysis on a Hollerith Tabulating Machine. Then moved to English Electric. Tried beta-reduction on the “Deuce” (successor to Pilot ACE)
- Aware of LISP from NPL Conference of Nov, 1958. Attended 1959 Copenhagen conference on Algol. Tutored Tony Hoare on Algol in 1961. Algol subcommittee in 1962.
- Employed by Christopher Strachey 1960-1964. Tasked with building a Mercury Autocode compiler for Ferranti computer. “Vaguely inspired” by early LISP. “Decided that the only way to do it was with the lambda calculus, of course; it was the only thing I knew”.

- Compiler work led to the SECD machine and its description in “The Mechanical Evaluation of Expressions” (1964). [10]
- R.C. Gilmore’s (1963) abstract machine for LISP exposed its “ramshackle semantic features”. Landin worked on “trying to put symmetry between treatment of functions and treatments of arguments”. It was “ill understood [at that time] that a function could be a data item”.
- In 1965, described semantics of parts of Algol 60 by mapping to (a syntactically sugared) lambda calculus. [7]
- In 1996, defined ISWIM, the core of a “family” of languages based on the lambda calculus.[4]

“Church without Lambda”

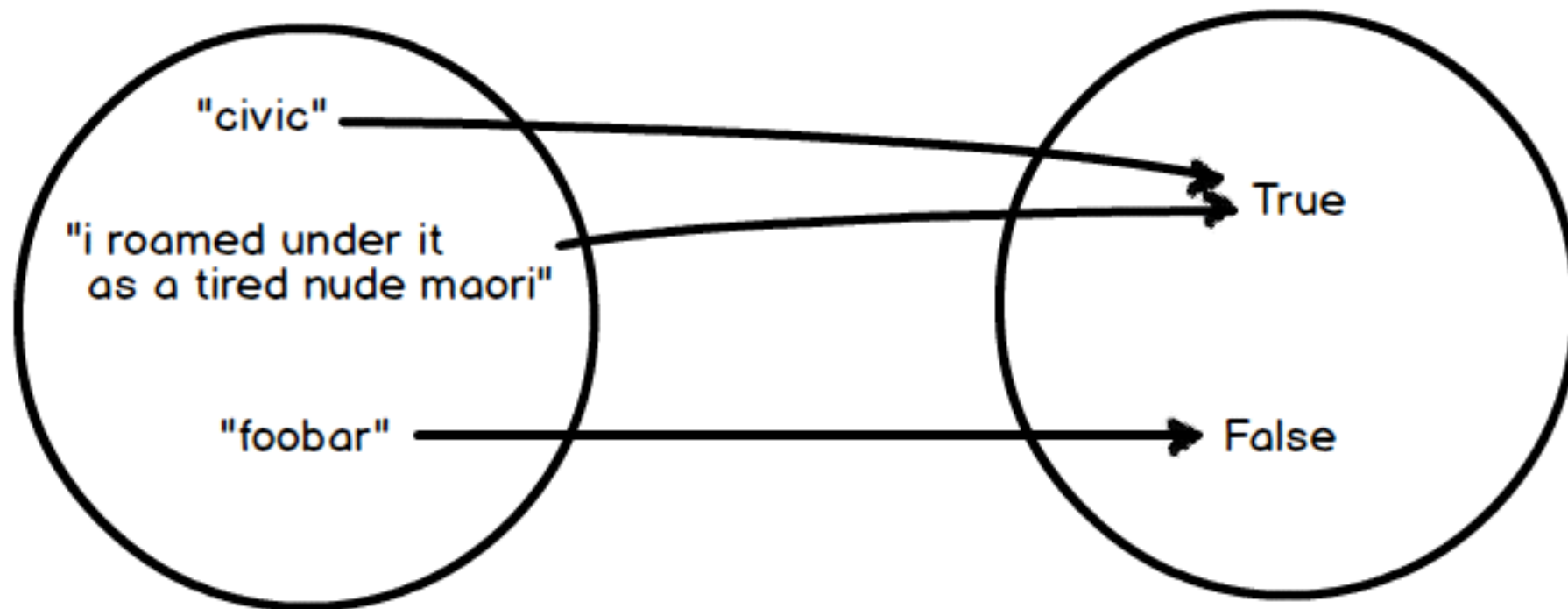
$(\lambda x. 10x^2 + 4x + 3) 5$

$10x^2 + 4x + 3$
where $x = 5$

let $x = 5$
 $10x^2 + 4x + 3$

properties of
domain?

properties of
co-domain?



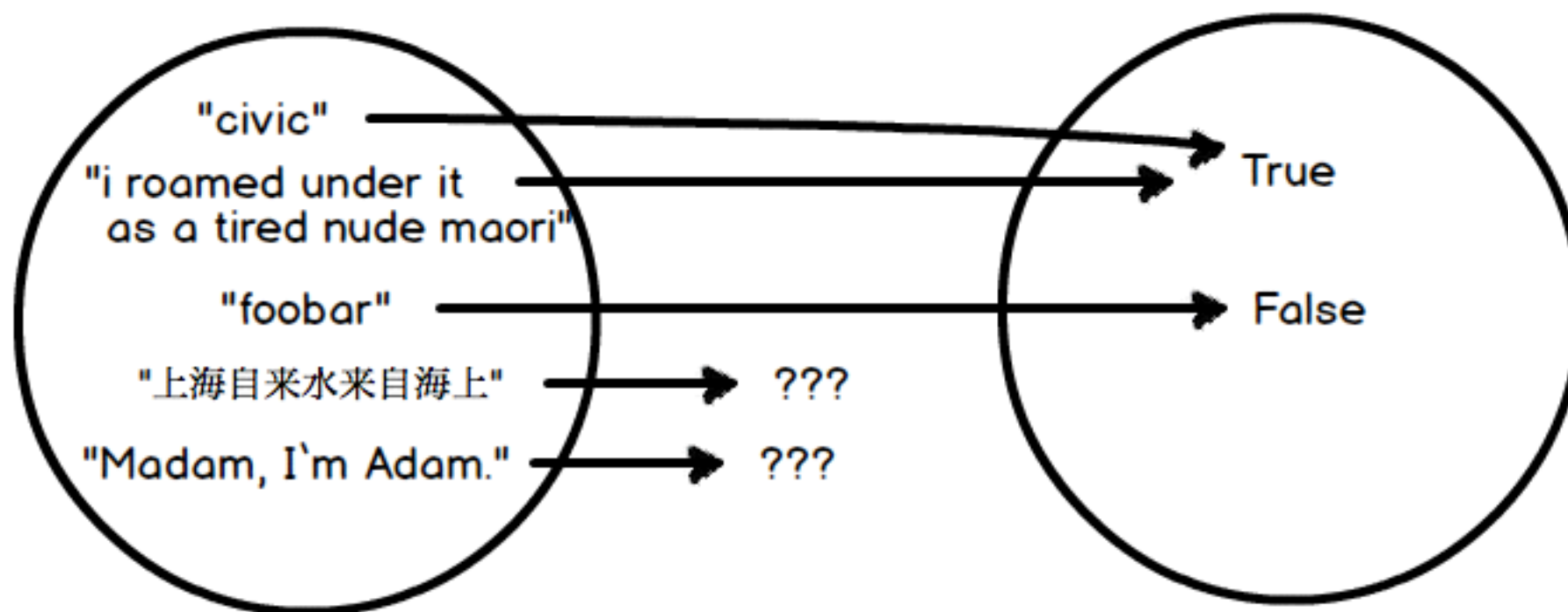
properties of
relation?

- “My real concern is with ... programs that are large due to the complexity of their task”
- “If a large program is a composition of N ‘program components’, the confidence level of the individual components must be exceptionally high if N is very large”

$$P = p^N \quad [0.99_1 * 0.99_2 * \dots * 0.99_{99} * 0.99_{100} = 0.37]$$

- “An assertion of program correctness is an assertion about the net effects of the computations that may be evoked”
- “Demonstration of correctness by sampling is completely out of the question”
- “The number of different inputs, i.e. the number of computations for which the assertions claim to hold is ... fantastically high” [8]

- *Edsger Dijkstra*
Structured Programming (1969)



- “Proof of program correctness should only depend upon the program text”
- “For what program structures can we give correct proofs without undue labor, even if the programs get large? how do we make, for a given task, such a well-structured program” [8]

- *Edsger Dijkstra*
Structured Programming (1969)

“The word "denotative" seems more appropriate than nonprocedural, declarative or functional. The antithesis of denotative is "imperative." Effectively "denotative" means "can be mapped into ISWIM without using jumping or assignment," given appropriate primitives.” [4]

- *Peter Landin*

The Next 700 Programming Languages (1966)

“ ... here are some of the steps by which many a conventional ALGOL60 or PL/1 program can be transformed into an ISWIM program that exploits ISWIM's nonimperative features. ...

(a) Replace a string of independent assignments by one multiple assignment.

(b) Replace an assignment having purely local significance by a where-clause.

(c) Replace procedures by type-procedures (possibly with multiple type), and procedure statements by assignment statements.

(d) Replace conditional jumps by conditional statements having bigger arms.

(e) Replace a branch whose arms have assignees in common by an assignment with conditional right-hand side.

(f) Replace a join by two calls for a procedure.” [4]

- *Peter Landin*

The Next 700 Programming Languages (1966)

```
def calcPremium(gender: String, tickets: Int, age: Int): Int = {  
    var premium = 0  
    if ( gender == 'M' ) {  
        if ( age < 21 ) {  
            premium = 1500 + 200 * tickets  
        } else if ( age >= 21 && age < 30 ) {  
            premium = 1200 + 100 * tickets  
        } else {  
            premium = 1000 + 100 * tickets  
        }  
    } else {  
        if ( age < 21 ) {  
            premium = 1200 + 200 * tickets  
        } else {  
            premium = 1000 + 100 * tickets  
        }  
    }  
    return premium  
}
```

```
def calcPremium(gender: String, tickets: Int, age: Int): Int = {  
  var premium = 0  
  if ( gender == 'M' ) {  
    if ( age < 21 ) {  
      premium = 1500 + 200 * tickets  
    } else if ( age >= 21 && age < 30 ) {  
      premium = 1200 + 100 * tickets  
    } else {  
      premium = 1000 + 100 * tickets  
    }  
  } else {  
    if ( age < 21 ) {  
      premium = 1200 + 200 * tickets  
    } else {  
      premium = 1000 + 100 * tickets  
    }  
  }  
  return premium  
}
```



←
←
← “my colleagues”

← “me”

← “my colleague”

```
def calcPremium(gender: String, tickets: Int, age: Int): Int = {  
  if ( gender == 'M' ) {  
    if ( age < 21 ) return 1500 + 200 * tickets  
    else if ( age >= 21 && age < 30 ) return 1200 + 100 * tickets  
    else return 1000 + 100 * tickets  
  } else {  
    if ( age < 21 ) return 1200 + 200 * tickets  
    else return 1000 + 100 * tickets  
  }  
  // Note that there's no further computation down here  
}
```

```
def calcPremium(gender: String, tickets: Int, age: Int): Int = {  
  if ( gender == 'M' ) {  
    if ( age < 21 ) 1500 + 200 * tickets  
    else if ( age >= 21 && age < 30 ) 1200 + 100 * tickets  
    else 1000 + 100 * tickets  
  } else {  
    if ( age < 21 ) 1200 + 200 * tickets  
    else 1000 + 100 * tickets  
  }  
  // Note that there's no further computation down here  
}
```

```
def calcPremium(gender: String, tickets: Int, age: Int): Int = {  
  if ( gender == 'M' ) {  meaningful equality?  
    if ( age < 21 ) 1500 + 200 * tickets  
    else if ( age >= 21 && age < 30 ) 1200 + 100 * tickets  
    else 1000 + 100 * tickets  
  } else {  still a “colleague”  
    if ( age < 21 ) 1200 + 200 * tickets  
    else 1000 + 100 * tickets  
  }  
}
```

"I propose that we abbreviate

if e **is** c_1 **then let** $c_1(x_1, \dots, x_{k1}) = e; \Phi_1(x_1, \dots, x_{k1})$
else if e **is** c_2 **then let** $c_2(x_1, \dots, x_{k2}) = e; \Phi_2(x_1, \dots, x_{k2})$
.....
else if e **is** c_n **then let** $c_n(x_1, \dots, x_{kn}) = e; \Phi_n(x_1, \dots, x_{kn})$

to

cases e :

$c_1(x_1, \dots, x_{k1}) = e; \Phi_1(x_1, \dots, x_{k1})$
 $c_2(x_1, \dots, x_{k2}) = e; \Phi_2(x_1, \dots, x_{k2})$
.....
 $c_n(x_1, \dots, x_{kn}) = e; \Phi_n(x_1, \dots, x_{kn})$

These case expressions were partly suggested by a case switch on type introduced into CPL by M. Richard" [9]

- Rod Burstall

Proving properties of programs by structural induction (1968)

“Lists can be characterized by a structure definition as follows:

A *list* is either *null*

or else has a *head* (h)

and a *tail* (t) which is a *list*.

” [10]

- *Peter Landin*

The Mechanical Evaluation of Expressions (1964)


```
data Gender = Man | Woman
```

```
calcPremium :: Gender -> Int -> Int -> Int
```

```
calcPremium gen age tkts = case (gen,age) of  
    (Man, age) | age < 21 -> 1500 + 200 * tkts  
                | elem age [21..30] -> 1200 + 100 * tkts  
                | otherwise -> 1000 + 100 * tkts  
    (Woman,age) | age < 21 -> 1200 + 200 * tkts  
                | otherwise -> 1000 + 100 * tkts
```



```
def getOdds(arr: Array[Int]): collection.mutable.ArrayBuffer[Int] = {  
    val arr2 = new collection.mutable.ArrayBuffer[Int]  
    for ( i <- ( 0 until arr.length)) {  
        if ((arr(i) % 2) == 1) {  
            arr2 += arr(i)  
        }  
    }  
    return arr2  
}
```

← irrelevant range

```
filter p [] = []  
filter p (x:xs) | p x = x: filter p xs  
                | otherwise = filter p xs
```

```
getOdds xs = filter (\n -> n `mod` 2 == 1) xs
```

Afterword

- Didn't cover the third formation rule of Böhm-Jacopini - Composition
- Loops have started to be abandoned. But conditionals using predicates on "Von Neumann" types persist.

Additional Resources

- Daniel Spiewak, “May Your Data Ever Be Coherent” at NEScala 2014
http://www.youtube.com/watch?v=gVXt1RG_yN0
- Bob Harper, “Boolean Blindness”
<http://existentialtype.wordpress.com/2011/03/15/boolean-blindness/>
- Tony Hoare, “Null References: The Billion Dollar Mistake”
<http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>

1. P. Naur, BIT 3(3): 204-205, 1963. Quoted in “DATALOGY - THE COPENHAGEN TRADITION OF COMPUTER SCIENCE” <http://www.naur.com/Datalogy-Naur.pdf>
2. C. Böhm and G. Jacopini, “Flow Diagrams, Turing Machines, and Languages With Only Two Formation Rules”, CACM Vol 9, No. 6, May 1966
<http://www.cs.unibo.it/~martini/PP/bohm-jac.pdf>
3. D. Knuth, “Structured Programming with go to Statements”, Computing Surveys, Vol. 6, No. 4, December 1974
<http://cs.sjsu.edu/~mak/CS185C/KnuthStructuredProgrammingGoTo.pdf>
4. P. Landin, “The Next 700 Programming Languages”, CACM Vol 9, No. 3, March, 1966
<http://www.cs.cmu.edu/~crary/819-f09/Landin66.pdf>
5. C. Strachey, “Fundamental Concepts in Programming Languages” Unpublished Lecture Notes for International Summer School in Computer Programming, Copenhagen, 1967
Reprinted in “Higher-Order and Symbolic Computation” 13:7-9, 2000
<https://www.itu.dk/courses/BPRD/E2009/fundamental-1967.pdf>

6. J. Backus, “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs”, CACM Vol 21, No. 8, August 1978
http://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf
7. P. Landin, “A Correspondence Between ALGOL 60 and Church’s Lambda-Notation”, CACM, Vol 8, No. 2, February 1969
<http://www.iro.umontreal.ca/~feeley/cours/ift6232/doc/a-correspondence-between-algol-60-and-churchs-lambda-notation.pdf>
8. E. Dijkstra, “Structured Programming”, Software Engineering Techniques, April 1970, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969
<http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF>
9. R. Burstall, “Proving properties of programs by structural induction”, The Computer Journal 12(1):41–48 (1969)
<http://www.cs.unibo.it/~martini/PP/bohm-jac.pdf>
10. P. Landin, “The mechanical evaluation of expressions”, The Computer Journal (1964) 6 (4): 308-320
<http://www.cs.cmu.edu/~crary/819-f09/Landin64.pdf>