Logo

Jamie Samdal

Ricardo Carlos

What is Logo?

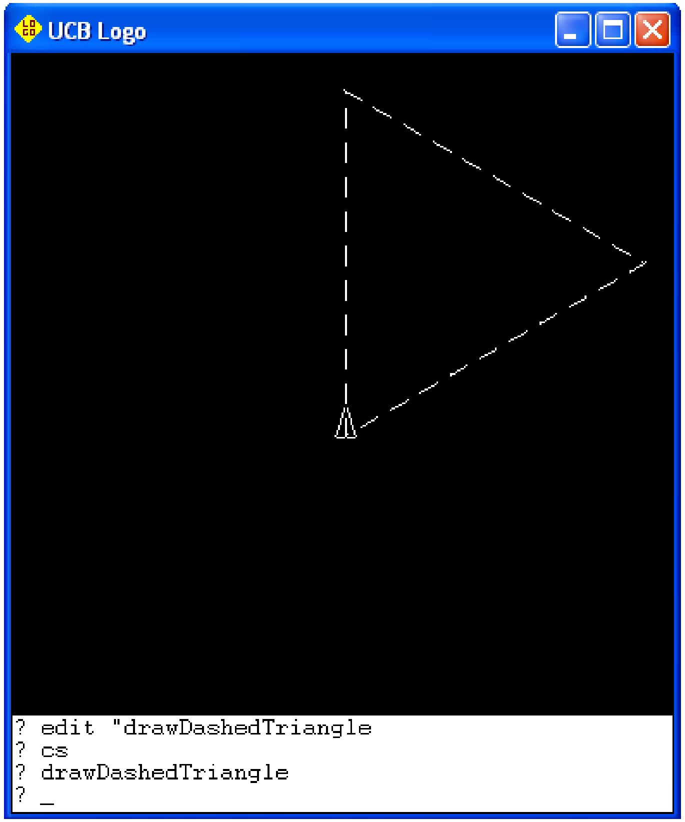
- Educational Language
  - Beyond Programming
  - Mathematics, Language, Music, Robotics, Science
- Novice Programmers
  - Interactive Interpreter
  - One Laptop Per Child
- Best Known For
  - Turtle Graphics
  - List and Language Processing

History

- First Version Created in 1967
  - Seymour Papert and Wallace Feurzeig
  - LISP
  - Introduce Children to Words and Sentences
- First Turtle Robot at MIT in 1969
- Over 170 Implementations in 2007
  - UCBLogo (Berkeley Logo)
  - Lego Logo

Turtle Graphics

```
to drawDashedTriangle
  repeat 3 ~
     [repeat 10 ~
          [forward 10 ~
           penup ~
           forward 10 ~
           pendown] ~
     right 120]
end
```

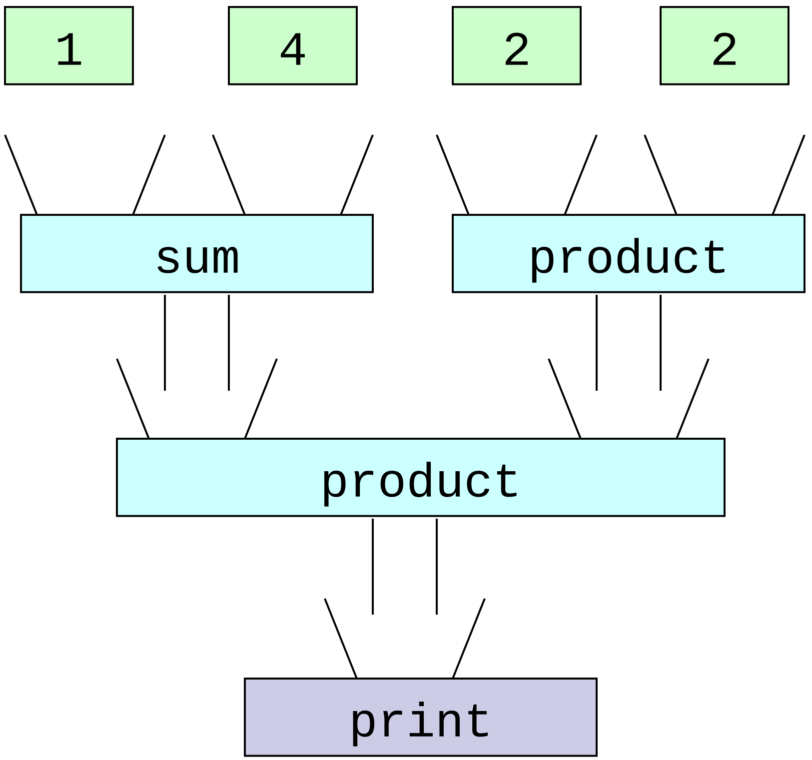


Types

- Word
  - Sequence of letters, numbers and punctuation
  - `word` is evaluated (variable or procedure)
  - `"word` is treated as a string
  - Numbers special case of word
- List
  - Contains words and lists, cannot modify after creation
  - `[one [2 3] four [five!]]`
- Array
  - Allows modification of a single element with setitem
  - `make "ar {one [2 3] four [[five]]}`
  - `setitem 4 :ar [five]`

Logo Instruction Evaluation

- Operation
  - `sum 3 4`
- Command
  - `print 20`
- Instruction
  - `print product sum 1 4 product 2 2`
- Errors
  - "Not enough inputs to print"
  - "You didn't say what to do with 2"



Example Logo Program

```
to switch :input :word1 :word2
  if equalp input [] [output []]
  if equalp :word1 first :input ~
    [output sentence :word2 ~
      switch butfirst :input]
  if equalp :word2 first :input ~
    [output sentence :word1 ~
      switch butfirst :input]
  output sentence first :input ~
    switch butfirst :input
end
```

Variables

- Variable: `var`
  - Quoted word is name
  - `thing "var` accesses the value (colon is shorthand for thing “var)
  - `make` changes the thing (value)
- `demonstrate`
  - Result: `x x var x`
  - var is local scope
  - y is global scope

```
to demonstrate
  local "var
  make "var "x
  print thing "var
  print :var
  make "y "var
  print :y
  make "y var
  print :y
end
```

Dynamic Scope

- Not declare before use
  - Assumed global scope
- Dynamic Scope
  - Scope of variables extends to called functions
- `caller "a "b`
  - Result: `a b a c`

```
to caller :a :b
  print :a
  print :b
  callee "c
end
to callee :b
  print :a
  print :b
end
```

Extensibility

- Users allowed to extend the language
  - Create new predicates
  - Create new control structures
- Act as both operations and commands
  - `ifelse` as operation: return a value

```
print ifelse emptyp [] [sum 2 3] [sum 6 7]
```

  - `ifelse` as command: execute other procedures

```
ifelse 4 = 2 + 2 [print "Y] [print "N]
```

Functional Programming

- Different paradigm from sequential
  - More focus on recursion instead of iterations
  - Combine sub-problem solutions to solve complex problems
    - Words created, then combined into sentences
- Logo: compromise between functional and sequential
  - Allows assignment and mutation
  - Turtle graphics programs use sequential programming

Summary

- Educational language founded in 1967
  - LISP derivative
  - Turtle graphics and language processing
  - Interactive interpreter
- Flexible and extensible
  - Variables: dynamic scoping
- Functional programming
  - More focus on recursion

CSc 520
Principles of Programming Languages

SETL

Pooja Bhandari, Tapasya Patki
Department of Computer Science
University of Arizona

And a SETL Program Looks Like…

SETS

MAPS

QUANTIFIERS

TUPLES

OPERATORS

```
program MinimumSpanningTree;

var V1 := { "A", "B", "C", "D", "E"};

var E1 := { [{"A","B"}, 13], [{"B","D"},7], [{"C","D"},2],
     [{"A","C"}, 2], [{"D","E"}, 1],[{"C","E"}, 4], [{"A","D"}, 5] };

MinST := PrimsAlgo(E1, V1);

procedure PrimsAlgo(mapOfEdges,setOfVertices);

  var nodes := { arb setOfVertices };
  setOfVertices := setOfVertices - nodes;
  while setOfVertices /= {} loop
    edges_selected := { [vertices,weight] in mapOfEdges |
                        (exists a in vertices | a in nodes) and
                        (exists b in vertices | b in setOfVertices) };

    [vertices,weight] := arb { [vertices,weight]  in edges_selected |
                  weight = min / { x : [-,x] in edges_selected }};

    node_selected := arb { t in vertices | t in setOfVertices };
    nodes with:= node_selected;
    setOfVertices less:= node_selected;
    spanningTree with:= [vertices,weight];

  end loop;
  return spanningTree;
end PrimsAlgo;

program MinimumSpanningTree;
```

History of the Language

- Designed by Jacob Schwartz at New York University in the 1970s
- To address set-intensive algorithms in compiler optimization
- Derived from Algol, APL, SNOBOL
- Syntactically similar to C, Perl
- Used for first Ada translator
- Dialects: SETL2, ISETL

When To Use SETL?

- Based on Mathematical Theory of Sets
- Very high-level
- Rapid Prototyping
- Supports transformational programming
- Translators, data processing systems, data-structure implementations, FOPL
- Slow compared to C
- Tradeoff: efficiency vs. expressiveness

Key Features

- At the nexus of imperative and declarative languages
- Wide-Spectrum
  - all levels of abstraction
- Weakly-typed
- Dynamically Typed
- Declaration-free
- Highly Orthogonal

- Automatic Memory Management
- Procedures
  - Pass-by-value
- Scope Rule
  - procedure scope
- Exception/Error Handling using `om`
- Value Semantics, and not pointer-based

Data Types and Operations

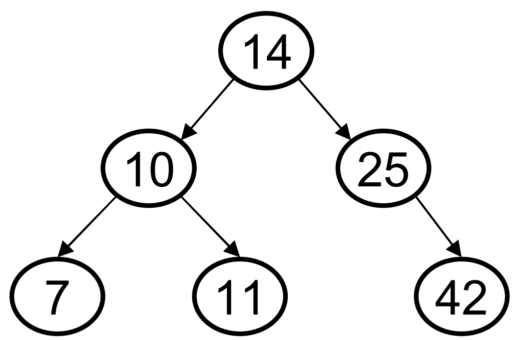
SETS	• Unordered • No duplicates
TUPLES	• Ordered • Duplicates possible • Can be extended dynamically • Allocated contiguous memory
MAPS	• Domain-Range mapping- sets of tuples of size 2 • Subset of a Cartesian Product • Single or Multi-Valued

Union
Intersection
Difference
Membership
Inclusion
Power Set
Domain
Range
Concatenation
Direct Retrieval

Data Structures Made Easy

```
Stack := [];
Stack with := 5;     $ Stack = [5]
Stack with := 7;     $ Stack = [5,7]
Data frome Stack; $ Stack = [5], Data = 7
Data frome Stack; $ Stack = om, Data = 5
```

TUPLES




```
BST := { [ 14, {10, 25} ], [ 10, {7, 11} ],
[ 25, {42} ] };
```

MAPS

Quantification, Assertions, Backtracking

- Compound Iterators
  - Forall
  - Exists
- Assertion
  - assert (*expr*)
  - signals an error if *expr* is false
- Backtracking
  - To explore other possible solutions

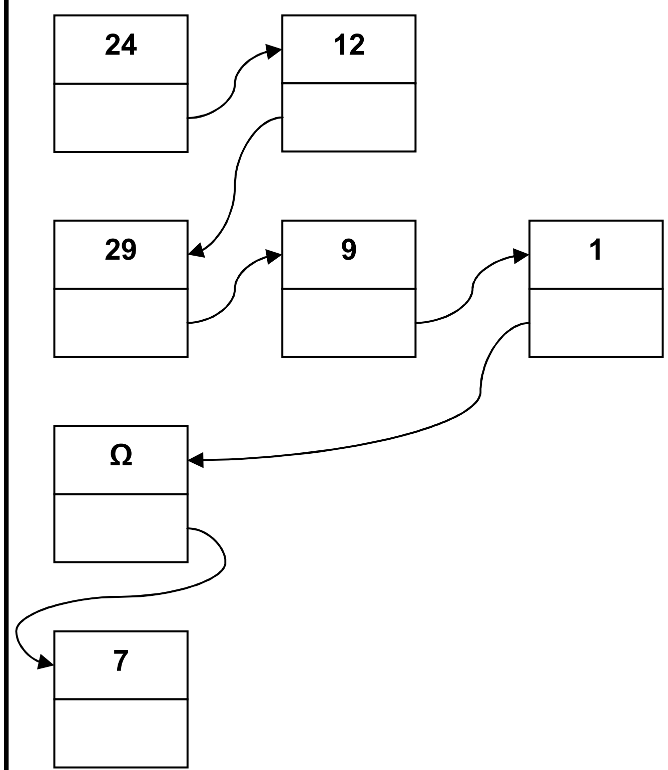
```
if forall x in t | x < 10
then
y := {3, 6, 9} ;
else y := {5, 10, 15};
end;
```

```
if exists a in setC | a > 10
then setC less := a;
end;
```

The Compiler's Perspective

Default Representation

- Doubly-Linked Hash Table (chained)
- Element blocks, linked together in a list to support iteration
- Indexed Vector Representation for Maps (facilitates sharing, less efficient)



x := { 29, 7, 24, 9, 1, 12 }

h (k) = k mod 4

Data Representation Sublanguage

- Code independent of data-representation
- Adds a system of declarations
- Supplemented Program
  - specify data structures and storage mechanisms for already written code
- Automatic DS Selection using ‘basing’

Consider a base *‘nodes’.* Then:
graph: **local map** (ϵ *nodes*) ϵ *nodes*;
path: **tuple** (ϵ *nodes*);

Garbage Collection

- i-Variable
  - Use to define mechanism
- o-Variable
  - Define to use mechanism
- Typically used Reference Counting
- Copy Collection has also been used for compiler optimization

SETL vs. Other Programming Languages

- Pascal
  - More expressive than Pascal
- Prolog
  - Declarative programming, backtracking
- Python
  - predecessor ABC was derived from SETL
- Java
  - SETL2 has packages and classes

Summary

- Expressive, very high-level
- Supports Rapid Prototyping
- Flexible, code is independent of data structure representation
- Value-based semantics
- Suitable for translator-design, transformational programming, proof-of-concept software design

Fortran 95

By: Eric Greene and Xing Qu

History of Fortran 95

- **Fortran: "*FOR*mula *TRAN*slating"**
- **Conceived by John Backus of IBM, 1953**
- **Fortran 95 program example**

```
program hello
   print *,"Hello 520!"
   print *,"Starting in 5"
   Do i=4, 1, -1
     print *,i
   ENDDO
end program hello
```

```
> f95 -o hello hello.f95
> hello
  Hello 520!
  Starting in 5
  4
  3
  2
  1
```

Target Audience

- Engineers, Mathematicians, Scientist
- For numerically intense programs,

  such as weather and climate modeling, computational tasks

Arrays

- Fortran Arrays require no specific index
  - The following arrays are all the same size:

```
INTEGER, DIMENTION(10) :: arr_1
INTEGER, DIMENTION(11:20) :: arr_2
INTEGER :: arr_3(-4:5)
```

  - Very beneficial to engineering and scientific applications

Arrays

- Array Manipulations are very sophisticated and quick in Fortran.
  - Large ranges can be changed without loops:

```
arr_1(5:9) = 3
```

  - Arrays can operate with arrays of like size:

```
arr_2 = arr_2 * arr_1              ! Multiply two arrays
arr_1(1:5) = arr_2(11:15) + 2  !  Change array part
arr_3(-4:5:2) = arr_1(6:10)     ! Stepped change
```

Arrays

- Fortran contains static, semi-dynamic, and dynamic arrays
  - Allocatable arrays are dynamic
  - Multidimensional arrays are native, cheap

- Variable-dimension array arguments can be passed through sub-routines

Parallelism

- Parallelism is a requirement for obtaining solutions for large and time consuming problems

- Procedural languages use a linear memory model, which make multiprocessing often impossible

- Fortran is the single exception

Parallelism

- Fortran supports implicit parallelism through array operations and syntax
- Usually limited to loops that satisfy a condition of data independence
- Just use –mp to compile a multiprocessor Fortran program!

Parallelism - Example

- Data Dependant code

```
Do i = 1,2
   A(i) = C(i)
   B(i) = A(i+1)
ENDDO
```

- Non-Parallel Execution

```
T1)  A(1) =C(1)
T2)  B(1) = A(2)
T3)  A(2) = C(2)
T4)  B(2) = A(3)
```

- In Parallel:

	Process #1	Process #2
T1)	A(1) =C(1)	A(2) = C(2)
T2)	B(1) = A(2)	B(2) = A(3)

- Parallel safe Fortran:

```
Do i = 1,2
   TEMP =  A (i+1)
   A (i) = C(i)
   B(i) = TEMP
ENDDO
```

Modularity

- In Fortran, Modules are used to group related procedures and data together
- Modules can be available in whole or part to other program units – easily portable
- Modularity brings OO conception into new-generation Fortran
  - Inheritance
  - Overloading
  - …

Modularity example

```
MODULE newbank
   use bank
   ! Variables in module
   private money
   public id

   ! functions or procedures
   interface Report
      module procedure Report_byID()
      module procedure Report_byName()
   end interface

contains
    subroutine Report_byID(num)
    …
    end subroutine Report_byID
```

```
subroutine Report_byName(name)
     …
end subroutine Report_byName
end module newbank


Program Main
  use newbank
  call SaveMoney(1000)
  call Report(collberg)
  call Report(12345)
end program Main
```

Summary

- **Fortran** is a general-purpose, procedural and imperative programming language.

- Which is especially suited to numeric computation and scientific computing.

- programs to benchmark and rank the worlds fastest supercomputers are written in Fortran.

-----http://en.wikipedia.org/wiki/Fortran

OCaml

By
Pavan Krishnamurthy
Qiyam Tung

OCaml vs JAVA

```
# let rec quicksort = function
   | [] -> []
   | pivot :: rest ->
       let is_less x = x < pivot
   in
       let left, right =
   List.partition is_less rest in
       quicksort left @ [pivot] @
   quicksort right
  ;;
```

```
public static void quicksort(double[]a)
    {shuffle(a);quicksort(a, 0, a.length -
    1); }
public static void quicksort(double[] a,
    int left, int right) { if (right <= left)
    return; int i = partition(a, left, right);
    quicksort(a, left, i-1); quicksort(a, i+1,
    right); }
 private static int partition(double[] a,
    int left, int right) { int i = left - 1;
    int j = right; while (true) { while
    (less(a[++i], a[right])); while
    (less(a[right], a[--j])) ; if (j == left)
    break; if (i >= j) break; exch(a, i, j); }
    exch(a, i, right); return i; }
private static boolean less(double x,
    double y) { comparisons++; return (x <
    y); }
private static void exch(double[] a, int i,
    int j) { exchanges++; double swap = a[i];
    a[i] = a[j]; a[j] = swap; }
private static void shuffle(double[] a)
    { int N = a.length; for (int i = 0; i < N;
    i++) { int r = i + (int) (Math.random() *
    (N-i)); exch(a, i, r); } }
```

History & Audience

- ML - Designed to develop theorem proving techniques.
- Designed by Robin Milner
- ML -> Caml -> Ocaml (Xavier Leroy 1996)
- Functional, but impure and fast
- Numerical programming and assisting in proofs
- Functional + Object-oriented

Aliases & Variants

```
let multiply x y = x * y;;
 let multiply2 = multiply 2;;
  multiply2 5;;
  -: int = 10
```

- Can define aliases for function names and arguments

```
type binary_tree = Leaf of
   int | Treeof
   binary_tree *
   binary_tree;;.
```

- Equivalent to unions in C, but cannot accidentally access wrong type

Pattern Matching

```
let rec mapNumToList =
  function
   | (_, []) -> []
   | (num, x::xs) ->
  num*x::mapNumToList
                (num,xs);;
```

- Simple and clean code easy to implement pattern matching
- Combined with variants, can be used to process symbolic algebra.

Type Inference & Parametric Polymorphism

```
class fun_point (y : int) =
   object
  val mutable x = y
  method get_x = x
  method set_x z =
  x <- z
  end;;

#let newfunc a = a#get_x;;
val newfunc : < die : 'a; ..
   > -> 'a = <fun>
#let p = new fun_point 7;;
#newfunc p;;
```

- Infers the type without need for explicit declaration
- Statically-typed + type inference avoids extra runtime checks
- Supports parametric polymorphism (generics)

Objects

```
 let maxmin x y =
     if x > y then object
   method max = x end
     else object method max =
   y end;;

(maxmin 3 4)#max;;
- : int = 4
```

- Objects similar to Java/C++ except one
- Supports Immediate objects

  (way to create an object directly instead of using class)

Functional Objects

```
#class fun_point y =
object
   val x = y
   method get_x = x
   method move d =
   {< x = x + d >} end;;

#let p = new fun_point 7;;
#p#get_x;;
   - : int = 7
#(p#move 3)#get_x;;
   - : int = 10
#p#get_x;;
   -:int =7
```

- {< … >} returns the object itself , a new object
- Orginial object is not altered

Garbage Collector

- Algorithm - Generational Collection

Summary

- OCaml is known for its superiority as type inferring and static type system language
- Noted for extending ML style programming to an object system
- It delivers atleast 50% of performance of a C compiler
- Compiler produces platform independent code

APL/J

Presented by

Qing Ju

Seung-jin Kim

THE UNIVERSITY OF ARIZONA®

JAVA VS J

Counting the number 99 in the given array (JAVA)

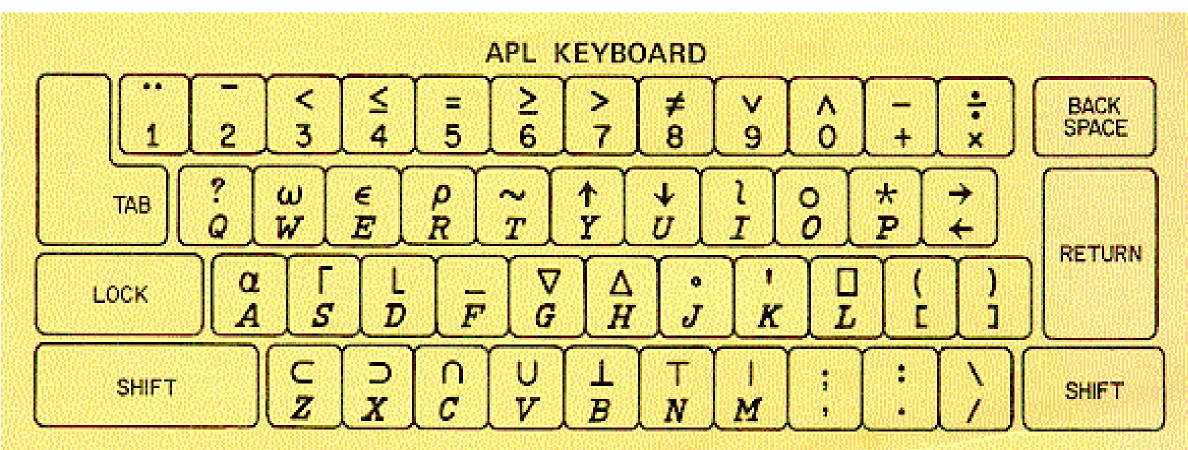
```
class count{
 public static void main(String args[]){
  int[] arr = {13, 45, 99, 23, 99};
  int count = 0;
  for (int i=0; i< arr.length; i++) {
   if( arr[i] == 99 ) count ++;
  }
  System.out.println(count);
 }
}
```

counting the number 99 in the given array (J)

```
+/99 = 23 45 99 23 99
```

History of APL

```
[6]     L←(L⍳':')↓L←,L
[7]     L←LJUST VTOM',',L
[8]     S←¯1++/∧\L≠'('
[9]     X←0⌈⌈/S
[10]    L←S⌽(-(⍴L)+0,X)↑L
[11]    A←((1↑⍴L),X)↑L
[12]    N←0 1↓DLTB(0,X)↓L
[13]    N←,'⍺',N
[14]    N[(N='_')/⍳⍴N]←' '
[15]    N←0 ¯1↓RJUST VTOM N
[16]    S←+/∧\' '≠⌽N
```



- A Program Language
- In 1957 by Kenneth E. Iverson
- Cryptic but powerful
- Array Programming Language

History of J

```
comb=: 4 : 0
 k=. i.>:d=.y-x
 z=. (d$<i.0 0),<i1 0
 for. i.x do. z=. k ,.&.> ,&.>/\. >:&.> z
end.
 ; z
)

cov=: 3 : 0 " 2
 r=. ,(|:y){"_1 M
 d=. N#. (*./"1@(e.&(i.N)) # ])
((#D)#y)+(y*&#D)$D
 I *./@e. r,d
)

qcover=: 4 : 0
 N=: y
 I=: i.N*N
 M=: (,: |:) i.N,N
 D=: 0 0 -.~ (,.~ , ] ,. |.)i: N-1
 (cov@((N,N)&#:) # ]) x comb N*N
)
```

- In 1990s by Ken Iverson and Roger Hui
- A successor to APL
- ASCII text
- No more mainframe
- Array Programing Language

Array processing in J

```
A =: 3 4 $ 1 4 8 2 4 2 4 3 4 5 4 3 4 2 3
A
   1 4 8 2
   4 2 4 3
   4 5 4 3
$A
   3 4
#A
   3
>: A
   2 5 9 3
   5 3 5 4
   5 6 5 4
```

Array processing in J

```
prices =: 40 15 20 40 41
orders =: 20 200 250 22 09

orders * prices
   returns 800 3000 5000 880 369
```

```
// Boxing

'good' ; 'morning'
+------+---------+
|good  |morning |
+------+---------+
```

Familiar Concepts in APL/J

- Dynamically Typing
- Pure FL (Referential Transparency)
- OO Features
- Garbage Collection
- Exception Handling

J: Creating Function

```
s =: - & 32
m =: * & (5%9)
convert =: m @: s
```

- Renaming   square =: *:
- Bonding   double =: *&2   tax =: 0.10&*
- Composing   (f @: g) y   means   f (g y)

  f and g are verb, and argument y

```
factorial =: */ @: >: @: i.
```

Fibonacci / Factorial f in J with OOP

```
    coclass 'MathFunctions' NB. 'coclass' define class
    fact =: 3 : 0
if. y < 1 do. 1
else. y*(fact y - 1)
end.
)
    fibo =: 3 : 0
if. y < 2 do. y
else. (fibo y-1) + fibo y-2
end.
)
    destory =: codestroy
    cocurrent 'base'            NB. End of class

M =: conew 'MathFunctions'    NB. Create new instance of class
   fact__M 3                  NB. call method fact in M
6
   fibo__M 10              NB. Call method fibo in M
55
```

Target audience

- Mathematical tools
- Engineering filed
- Prototype developing(SAP),
- Financial quantitative tools
- Educational uses

Conclusion

- Strong Array Processing
- FL Features
- OOP Features
- User-Friendly Environment

Erlang: Concurrency Oriented Programming

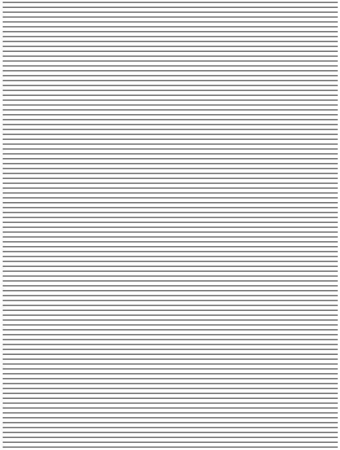
There and Back Again

R. Bailey   I. Ryan

Department of Computer Science
University of Arizona

5 May 2008 / CS520 Final Paper

Equivalent Portion in C++



Portion of EGGS

```
-module(eggs).
-export([ server_node/0, start_server/0, server/1, logon/1,
logoff/0, msg/1, msg/2, client/4 ]).
server_node() -> srvr.
start_server() ->
register(srvr, spawn(eggs, server, [ [] ])),
server_node() ! initiate.
logon(UserName) ->
server_node() !  self(), logon, UserName}.
logoff() ->
self() ! logoff.
msg(Text) ->
self() ! {message_send, all, Text}.
msg(UserName, Text) ->
self() ! {message_send, UserName, Text}.
```

- Created by Ericsson, the Swedish telecommunications company
- Needed high level symbolic language
- LISP, Prolog lacked concurrency primitives
- Released as open source in 1998

```
-module(hofstadter).
-export([hof/1]).
hof(1) -> 1;
hof(2) -> 1;
hof(N) when N > 2 ->
hof(N-hof(N-1))+
hof(N-hof(N-2)).
```

- Recursion-based iteration
- Similarities to Ada specifications

M = 3.14.
N = 5.
P = 'hello'.
N = 42. ← Error!

- Dynamic typing similar to Smalltalk
- Variables can't be changed

```
-module(mult). -export([mult/2,
mult/3]).
mult(X,Y) -> X * Y.
mult(J, K, L) -> J * K * L.
```

- Local function
- Overloading, but different number of arguments

'EXIT', From, Reason ->
io:format("eggs exiting, got p n", ['EXIT', From, Reason])

- Errors don't affect other processes
- Sends error message

```
start_server() ->
register(srvr, spawn(eggs,
server, [ [] ])),
server_node() ! initiate.
```

- Create and manage threads
- No shared state
- Great for webservers

```
receive
listusers ->
io:format("srvr: Users :  p  n",
[Userlist]), server( Userlist );
```

- Create finite number of new proccesses
- Pass finite number of messages
- Designate behavior for next message

?MODULE:codeswitch (server_node)

- Minimize application downtime
- Switch to different version already loaded
- Create new process and load into application

Summary

- Easy to program large applications
- Performance scales with number of processors
- Hot-swappable modules allow for high uptime