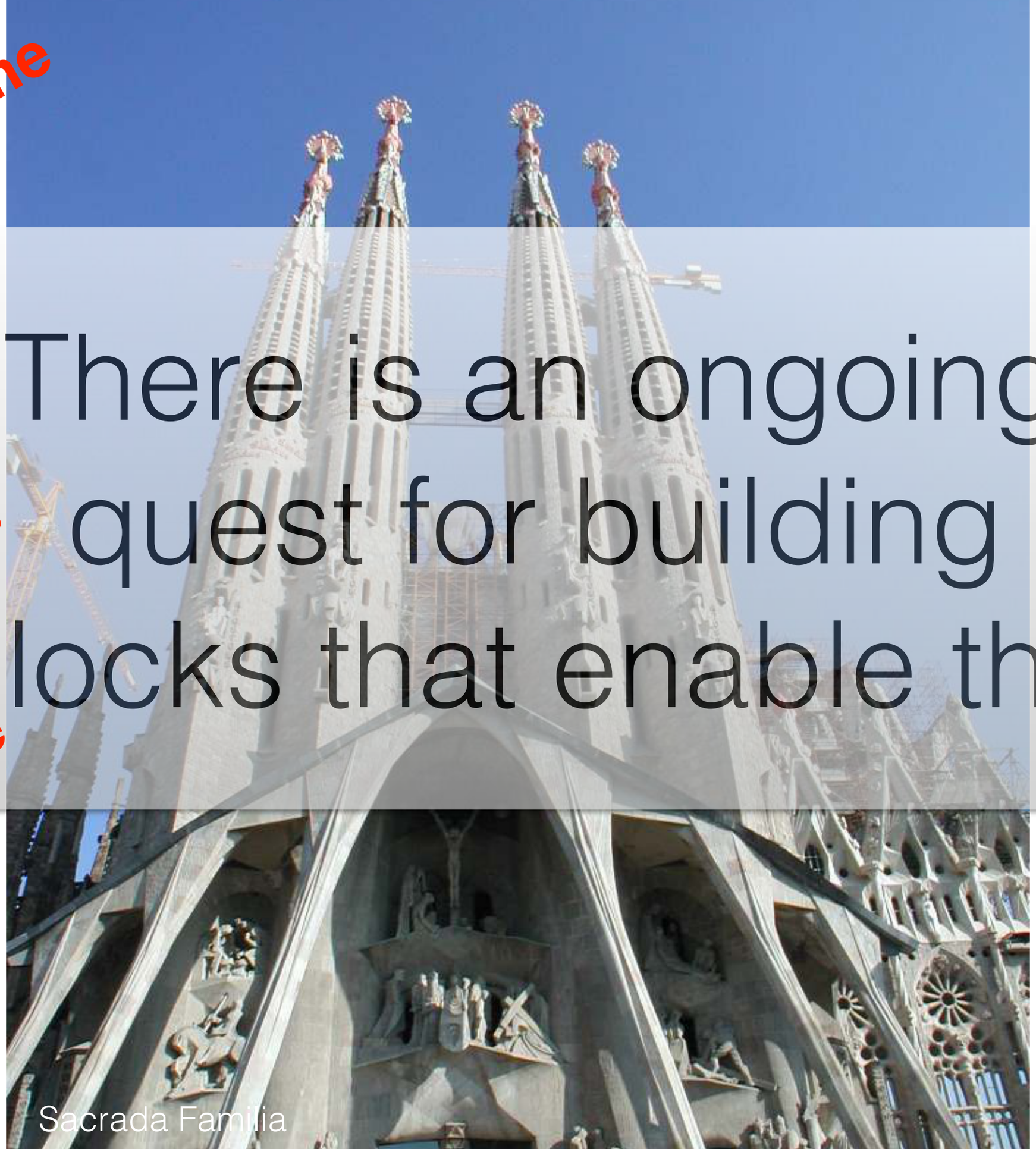
revenge of the coroutines

Paul Klint

CWI

Software Engineering is about designing and building **beautiful, possibly large software systems**

Johnson Wax Building Headquarters



There is an ongoing quest for building blocks that enable this

subroutine
actor
module
thread
process
fiber
package
service
class
bus

Sacrada Familia

I will explore the
coroutine
as building block
(in the context of building a Rascal compiler)

Joint work with

- Anastasia Izmaylova (coroutines)
- Mark Hills (type checking)

What is a coroutine?

Coroutines are computer program components that generalize subroutines for nonpreemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations.

Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, exceptions, event loop, iterators, infinite lists and pipes.

Source: 

Why?

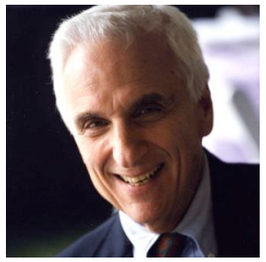
revenge

Maurice Wilkes

subroutine (1951)
control

David Wheeler

coroutine (1958)
data + control



Melvin Conway

Ole-Johan Dahl

Simula (1967)
data + control

Kristen Nygaard

Data

Control

C++ (1983)
Java (1995)

Object-oriented languages

Process-oriented languages

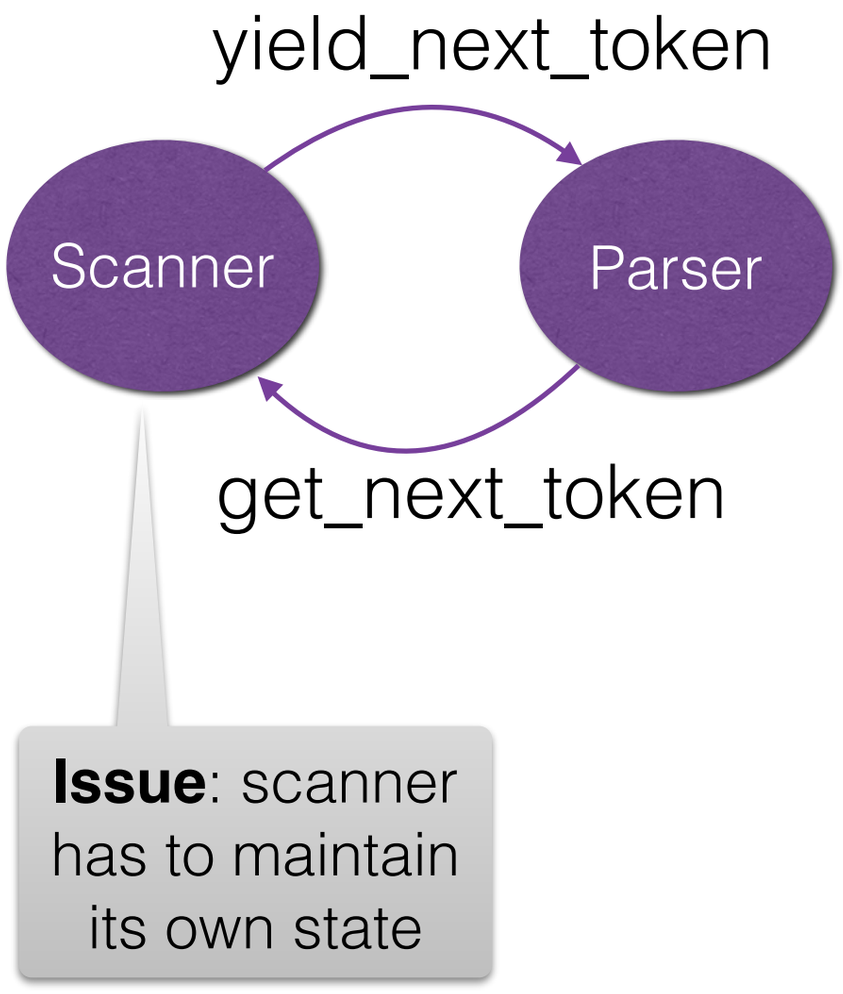
Occam (1983)
Erlang (1986)

Iterators
Greenlets
Fibers
Threads

Clu (1974)
Icon (1977)
C# (2000)

data + control!
(2014)

Conway, M. E. (July 1963). *Design of a Separable Transition-Diagram Compiler, CACM* **6** (7): 396–408.



Issue: scanner has to maintain its own state

Design of a Separable Transition-Diagram Compiler*

Melvin E. Conway
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) Cobol [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed Cobol compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective Cobol except for automatic segmentation and its byproducts, such as those properties of the alter verb which are affected by segmentation. The verbs define, enter, use and include are accessible to the design but were not included in the prototype coded at the Case Computing Center.
2. It can be implemented as a true one-pass compiler (with load-time fixup of forward references to procedure names) on a machine with 10,000 to 16,000 words of high-speed storage. In this configuration it processes a source deck as fast as current one-pass algebraic compilers.
3. It can be segmented into many possible configurations, depending on the source computer's storage size, such that (a) once a segment leaves high-speed storage it will not be recalled; (b) only two working tapes are required, and no tape sorting is needed. One such configuration requires five segments for a machine with 8000 six-bit characters of core storage.

Of course any compiler can be made *one-pass* if the high-speed storage of the source computer is plentiful enough; therefore, what this exposition has to offer is a collection of space-saving techniques whose benefits are real enough to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to Cobol compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete Cobol syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

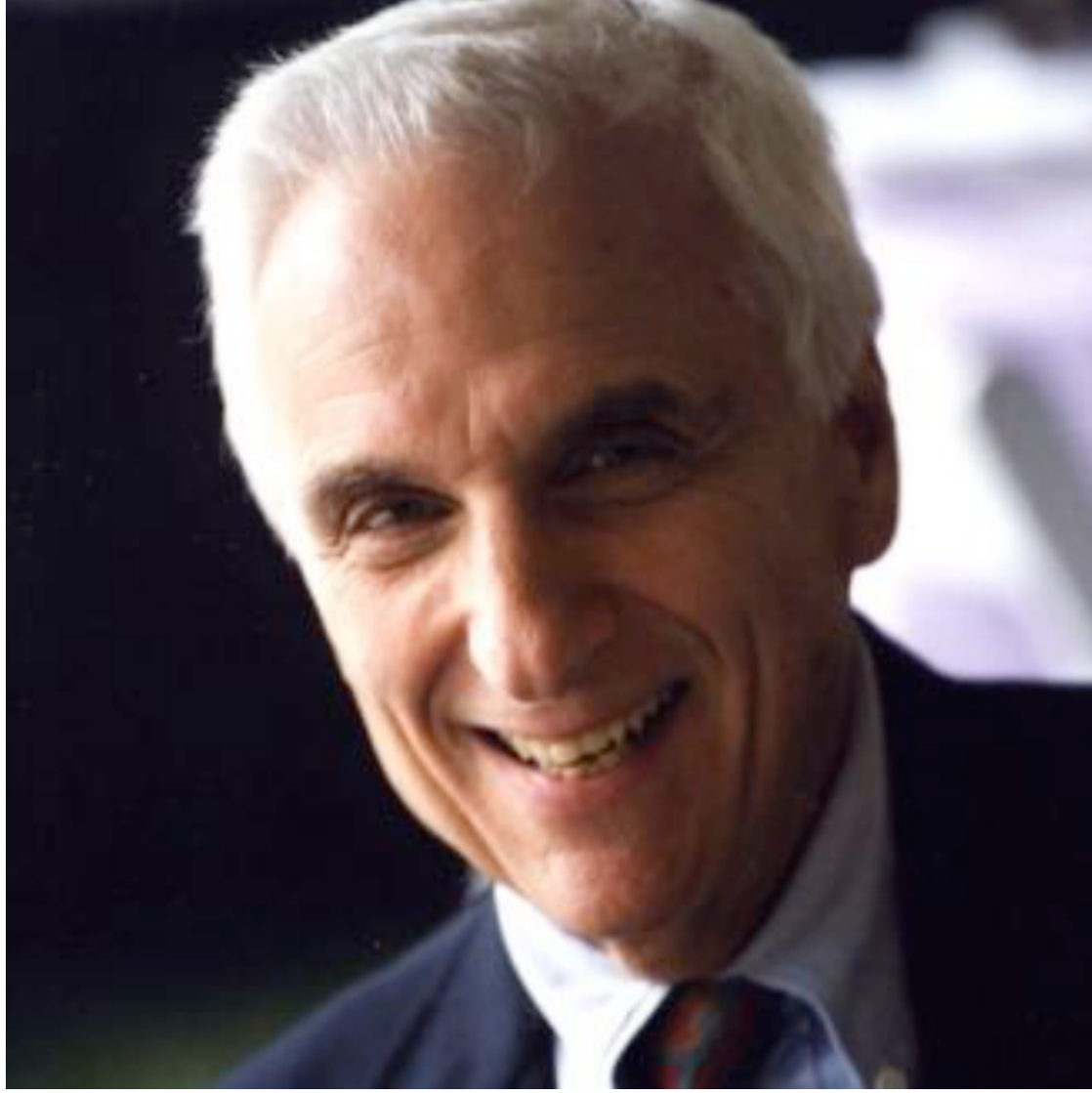
Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on the number of inputs and outputs a coroutine may have.

The coroutine notion can greatly simplify the conception of a program when its modules do not communicate with each other synchronously. Consider, as an example, a program which reads cards and writes the string of characters it finds, column 1 of card 1 to column 80 of card 1, then column 1 of card 2, and so on, with the following wrinkle: every time there are adjacent asterisks they will be paired off from the left and each "**" will be replaced by the single character "↑". This operation is done with the exponentiation operator of Fortran and Cobol. The flowchart of such a program, viewed as a subroutine pro-

* The work described here was performed at Case Institute of Technology in 1961 and was supported in part by Univac Division of Sperry Rand Corporation.

[1] To the best of the author's knowledge the coroutine idea was concurrently developed by him and Joel Erdwinn, now of Computer Sciences Corporation.

(aside)



Conway’s Law

"Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."

return *vs* yield

```
subroutine P(…){
  …
  return v;
  …
}
```

Code following the return will never be executed

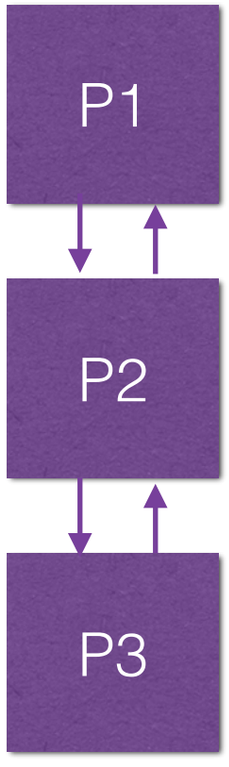
```
coroutine C(…){
  …
  yield v;
  …
}
```

Code following the yield will be executed when C is resumed

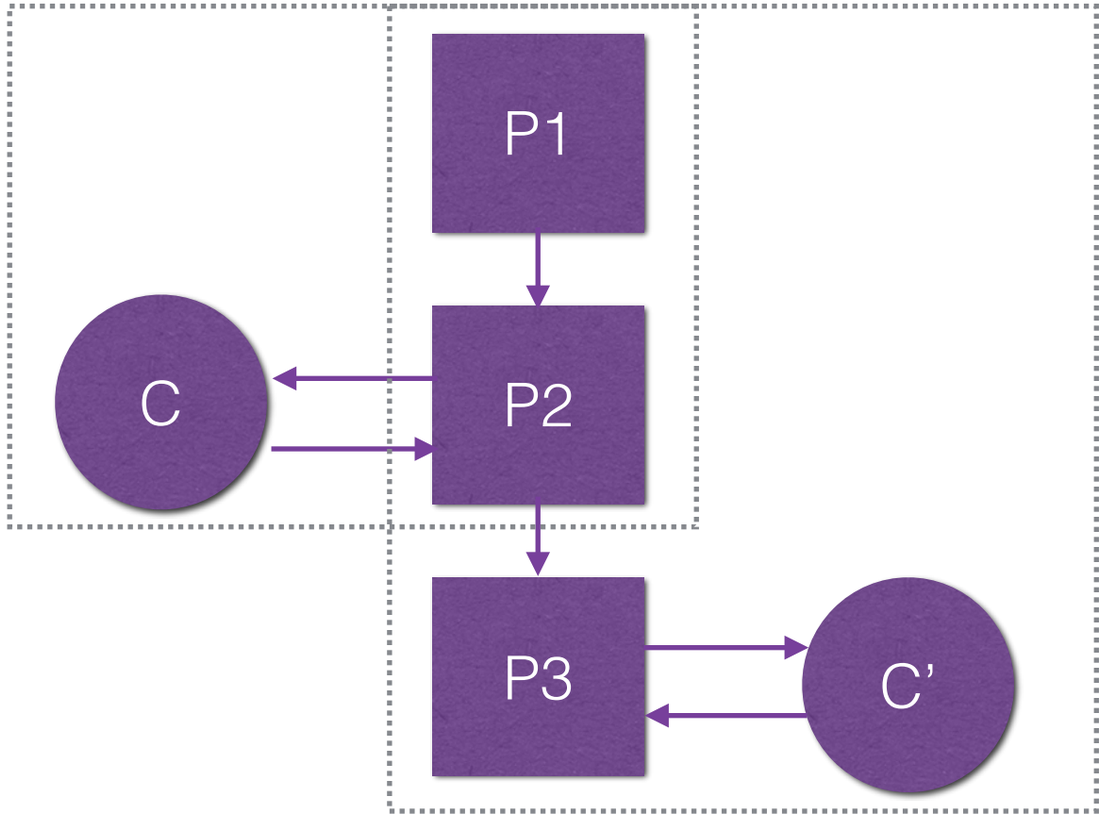
A subroutine is just a special case of a coroutine (Donald Knuth)

Consequence

- Procedures (and functions and methods) can be implemented using a single stack



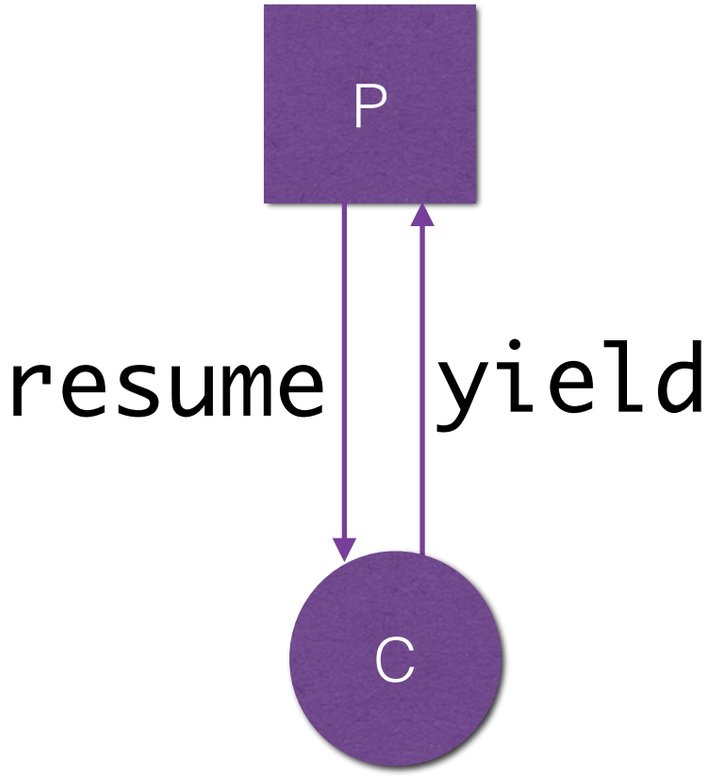
- Since coroutines can be resumed, their call stack needs to be retained (but common parts can be shared)



Flavors of Coroutines, 1

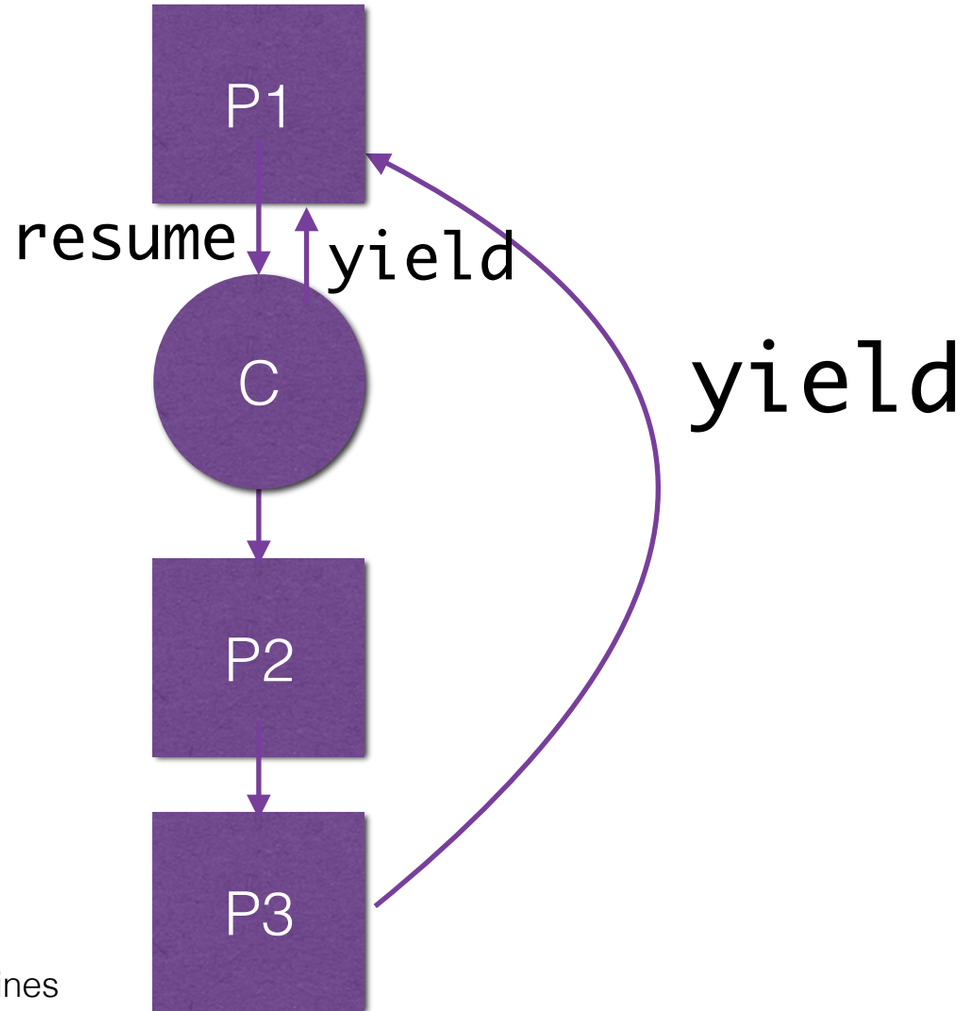
Stackless

- yield may only occur in the coroutine itself
- Typical use: Iterators



Stackfull

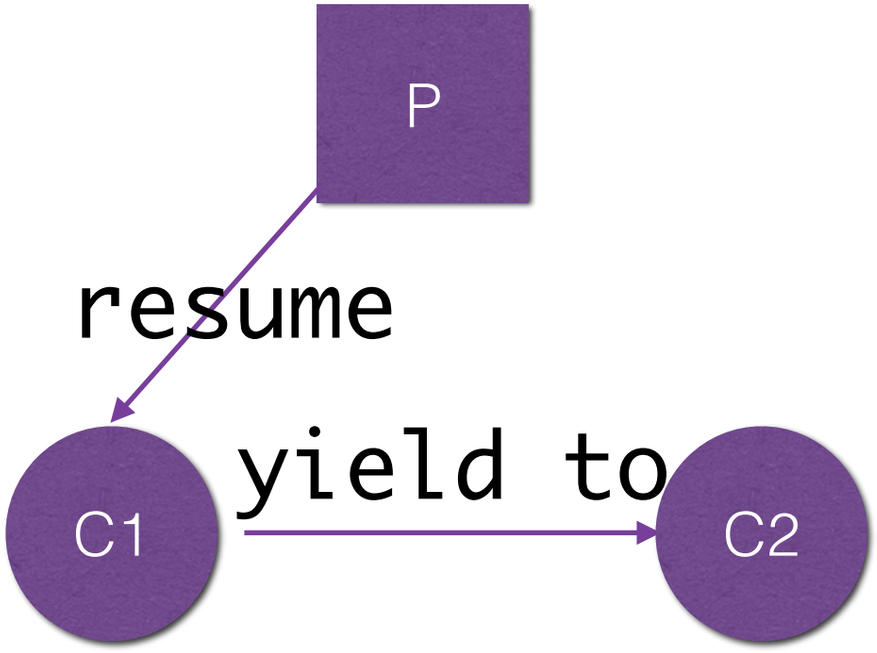
- yield may occur in called functions
- Typical use: explore complex search space, multi-tasking



Flavors of Coroutines,2

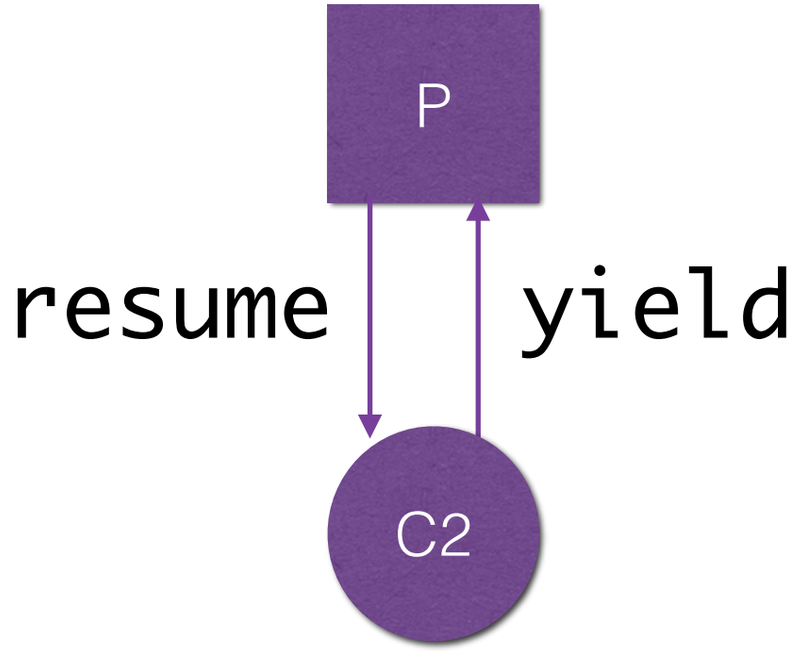
Symmetric

- a coroutines can yield to or resume any other coroutine
- Typical use: cooperative processes



Asymmetric

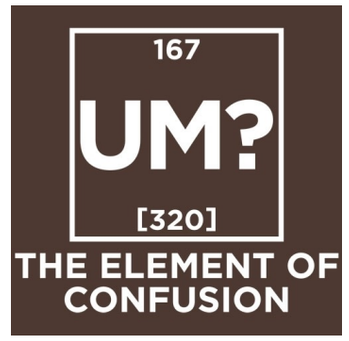
- a coroutine can only yield to its resumer
- Typical use: client/server, master/slave



Some Facts

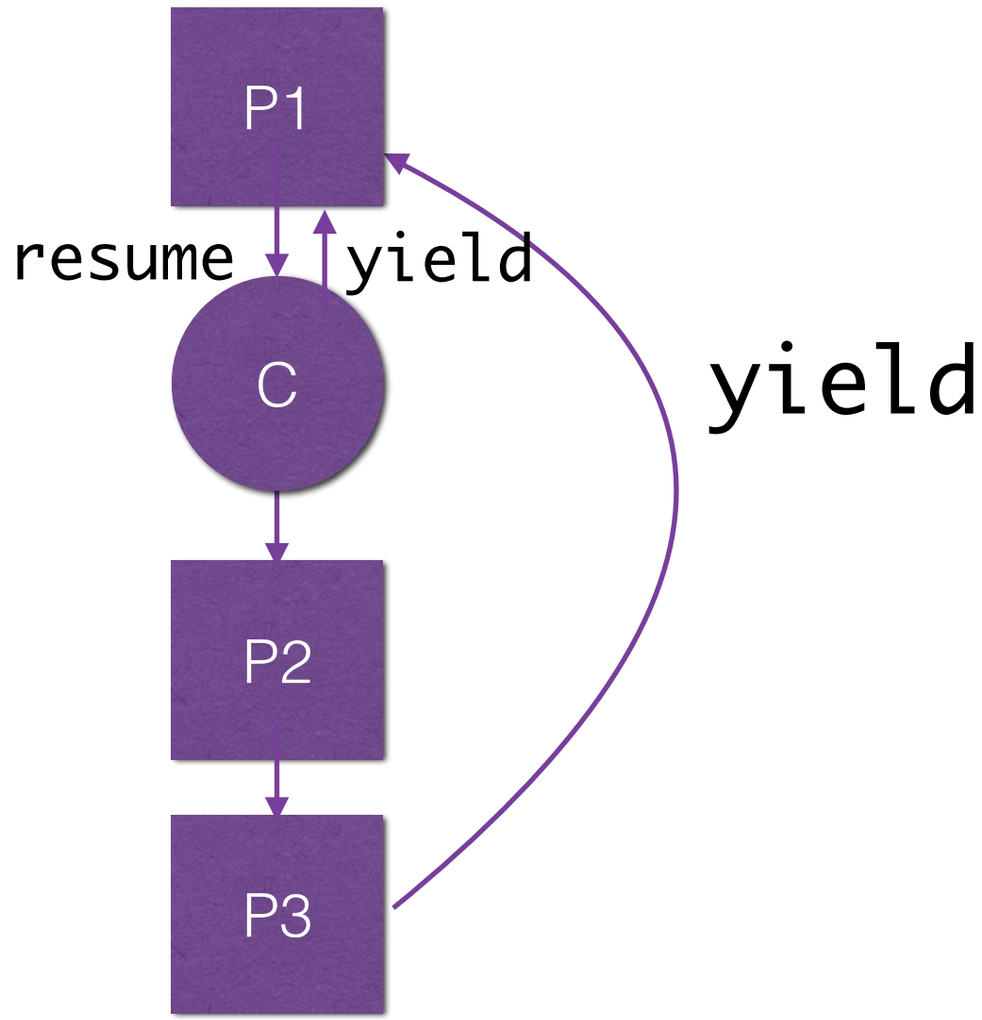
- Stackful coroutines have larger expressive power than stackless coroutines
- Symmetric and Asymmetric coroutines are equivalent
- Coroutines are equivalent to one-shot continuations
- *See*: A. De Moura & R. Ierusalimschy, Revisiting Coroutines, ACM TOPLAS, **31** (2) 2009, pp.1-31
- There is a **bewildering** amount of different names used for identical concepts. Makes it hard to compare languages!

Some languages that support coroutine-*like* functionality



- C# (iterators, fibers, stackless?)
- C++ (fibers=cooperative threads)
- D (fibers)
- Erlang (processes ~ fibers)
- F# (coroutines built on callcc)
- Haskell (implied by lazy eval)
- Icon (iterators, stackful)
- Javascript (generators, stackless)
- Julia (tasks, symmetric, stackless)
- Lua (asym, stackful coroutines)
- PHP (generators, stackless?)
- Python (generators, stackless)
- Ruby (generators and callcc)
- Scheme (coroutines built on callcc)
- Simula 67 (sym, stackful)

Focus now on **asymmetric**, **stackful** coroutines



Design considerations

How to signal termination?

- Use a `hasNext`, `next` interface
  - Java `Iterator` interface
- Use a `MoveNext`, `Current` interface
  - C# `Enumerable` interface
- Return a special value (null, false)
  - Mostly used in dynamically typed languages, e.g. Lua, Ruby
- Raise an exception: e.g., Lua, Python

Design considerations

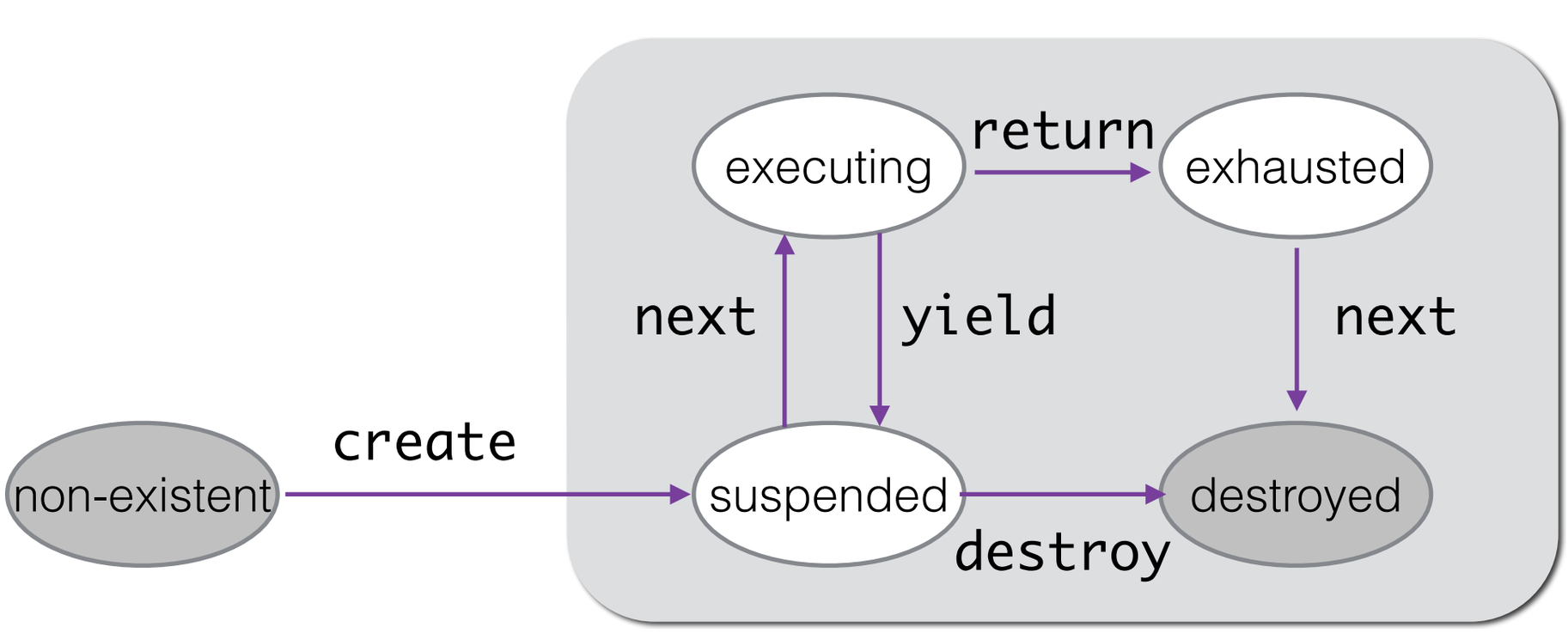
How to return a value?

- As return value of next/resume/Current…
  - Java, Ruby, Lua, …

- As a reference parameter of the coroutine (see muRascal, next slide)
  - Increased flexibility
  - Results can go to different scopes

Lifecycle of a muRascal coroutine

muRascal is a coroutine-based **intermediate** language used in the Rascal compiler



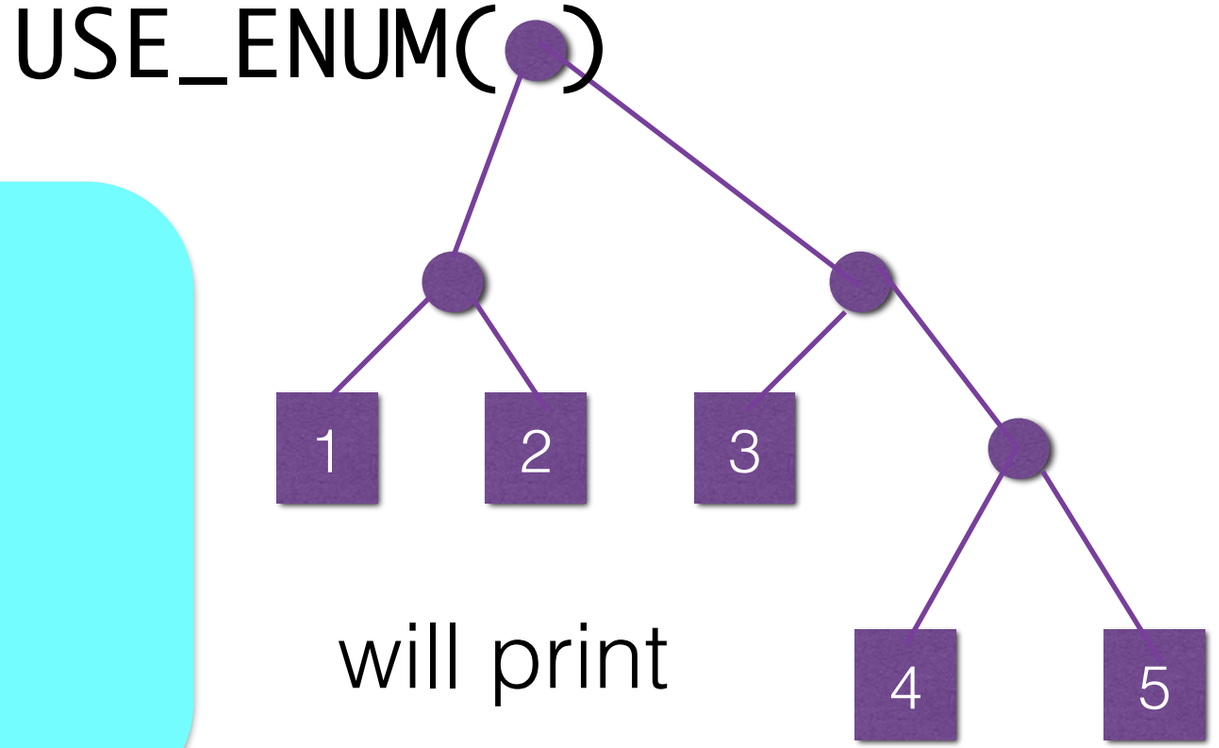
Tree traversal

Tree to be traversed

Reference parameter to which result is assigned

```
coroutine ENUM(tree, rVal){
  if(tree is Leaf)
    deref rVal = tree;
    yield
  else {
    ENUM(tree.left, rVal)
    ENUM(tree.right, rVal)
  }
}
```

```
function USE_ENUM(tree){
  var val,
      enum = create(ENUM, tree, ref val)
  while(next(enum))
     println(val)
}
```



will print

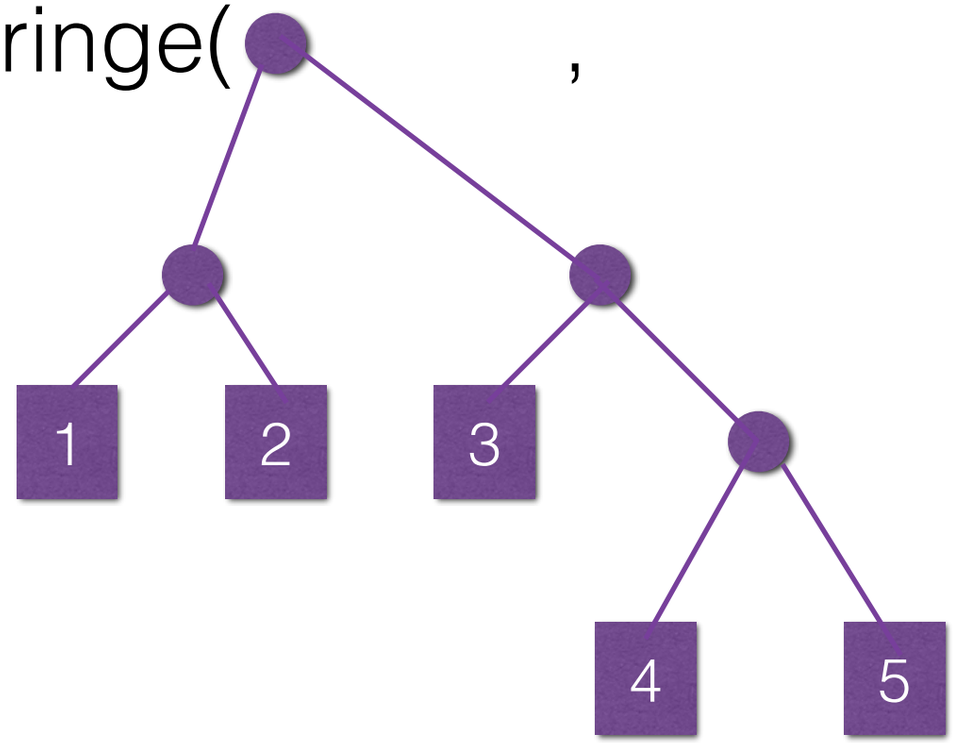
1
2
3
4
5

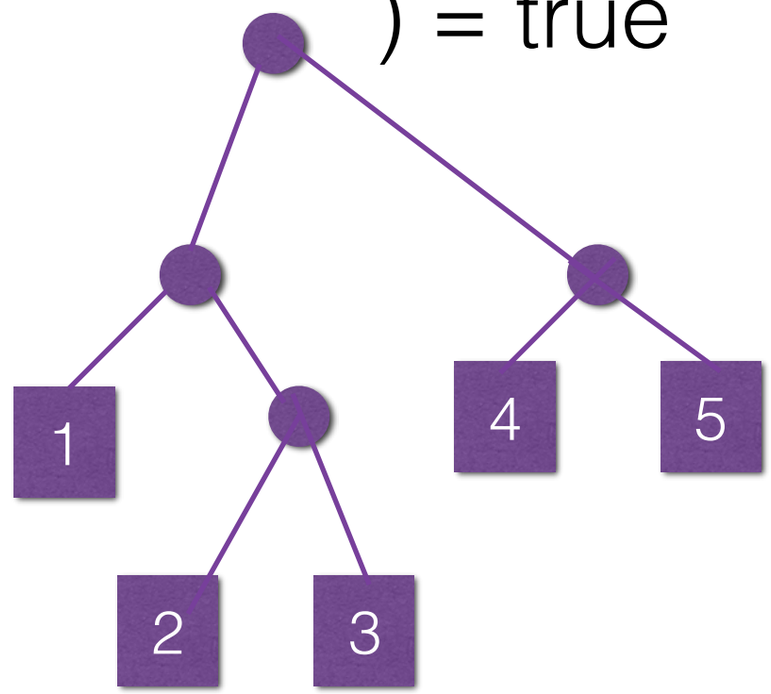
The Same Fringe Problem

- Given two trees determine whether they have the same leaves in left-to-right order

- **Constraint**: ***traverse each tree only once***

sameFringe(,) = true





sameFringe in Python

```
from itertools import izip_longest

def fringe(tree):
    """Yield tree members L-to-R depth first, as if stored in a binary tree"""
    for node1 in tree:
        if isinstance(node1, tuple):
            for node2 in fringe(node1):
                yield node2
        else:
            yield node1

def same_fringe(tree1, tree2):
    return all(node1 == node2
               for node1, node2 in izip_longest(fringe(tree1), fringe(tree2)))
```

sameFringe in Haskell

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
    deriving (Show, Eq)

fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Node n1 n2) = fringe n1 ++ fringe n2

sameFringe :: (Eq a) => Tree a -> Tree a -> Bool
sameFringe t1 t2 = fringe t1 == fringe t2
```

- Lazy evaluation gives simple solution for “traverse once” requirement

- **Coroutines bridge the gap between eager and lazy evaluation**

`sameFringe` in muRascal

As we have seen before:

```
coroutine ENUM(tree, rVal){
  if(tree is Leaf)
    deref rVal = tree; yield
  else {
    ENUM(tree.left, rVal)
    ENUM(tree.right, rVal)
  }
}
```

```
function SAME_FRINGE(tree1, tree2){
  var val1, val2
      enum1 = create(ENUM, tree1, ref val1)
      enum2 = create(ENUM, tree2, ref val2)
  while(next(enum1))
    if(!next(enum2) || val1 != val2)
      return false
  return !next(enum2)
}
```

Let’s apply muRascal’s coroutines …

Problem area: matching of complex data structures:

- Lists, sets, bags, …

In particular: how to efficiently compile such pattern matches?

List Matching Examples, 1

pattern

matches

subject

```
[1, 2, 3, 4, 5] := [1, 2, 3, 4, 5]
```

```
[1, 2, 3, 4, 7] := [1, 2, 3, 4, 5]
```

variable

```
[1, 2, 3, V, 5] := [1, 2, 3, 4, 5]
```

$V \mapsto 4$

List Matching Examples, 2

list variable

```
[1, *L, 5] := [1, 2, 3, 4, 5]
```

L ↦ [2,3,4]

variable

```
[1, *L, 4, V] := [1, 2, 3, 4, 5]
```

L ↦ [2,3]

V ↦ 5

List Matching Examples, 3

```
[1, *L, *M, 5] := [1, 2, 3, 4, 5]
```

L ↦ []
M ↦ [2,3,4]

L ↦ [2,3]
M ↦ [4]

L ↦ [2]
M ↦ [3,4]

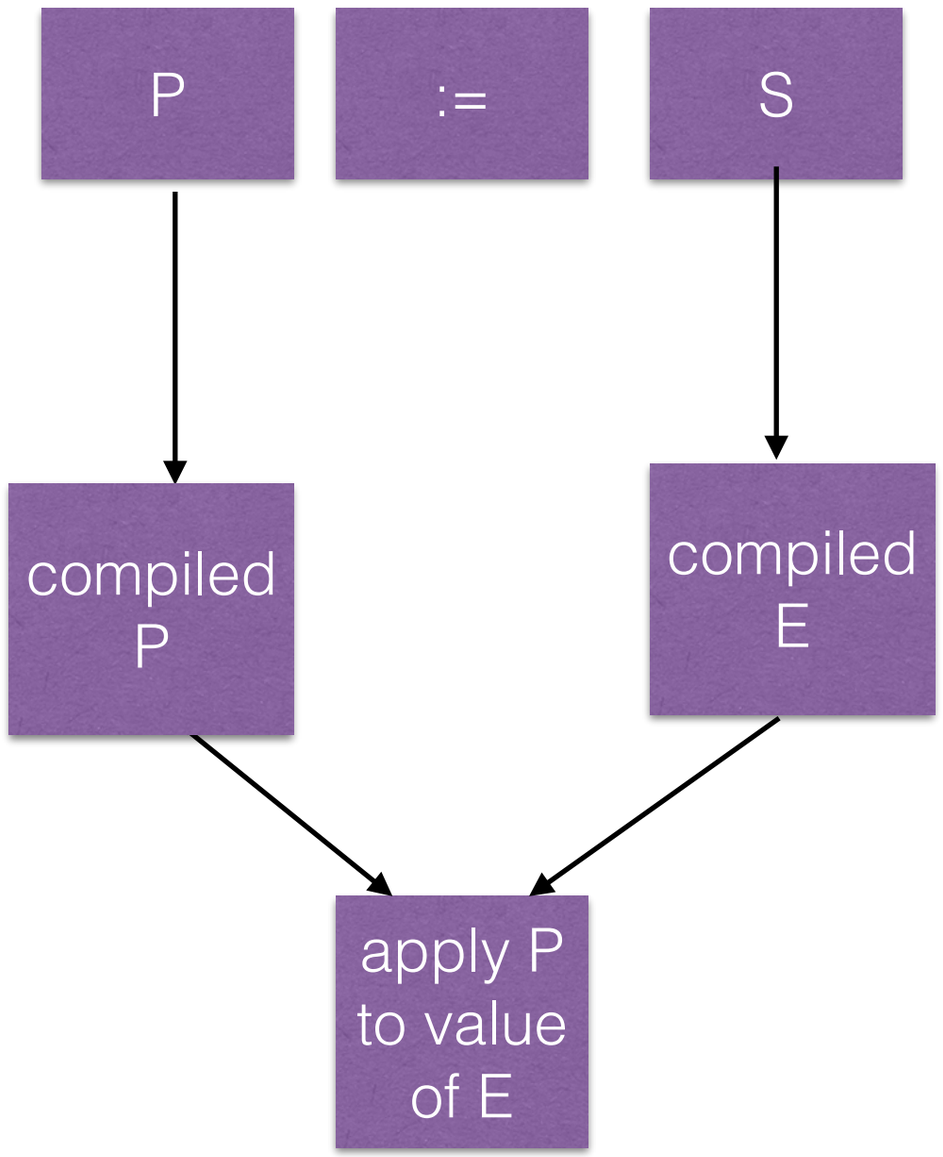
L ↦ [2,3,4]
M ↦ []

Multiple solutions with multiple variable bindings

Full Disclosure

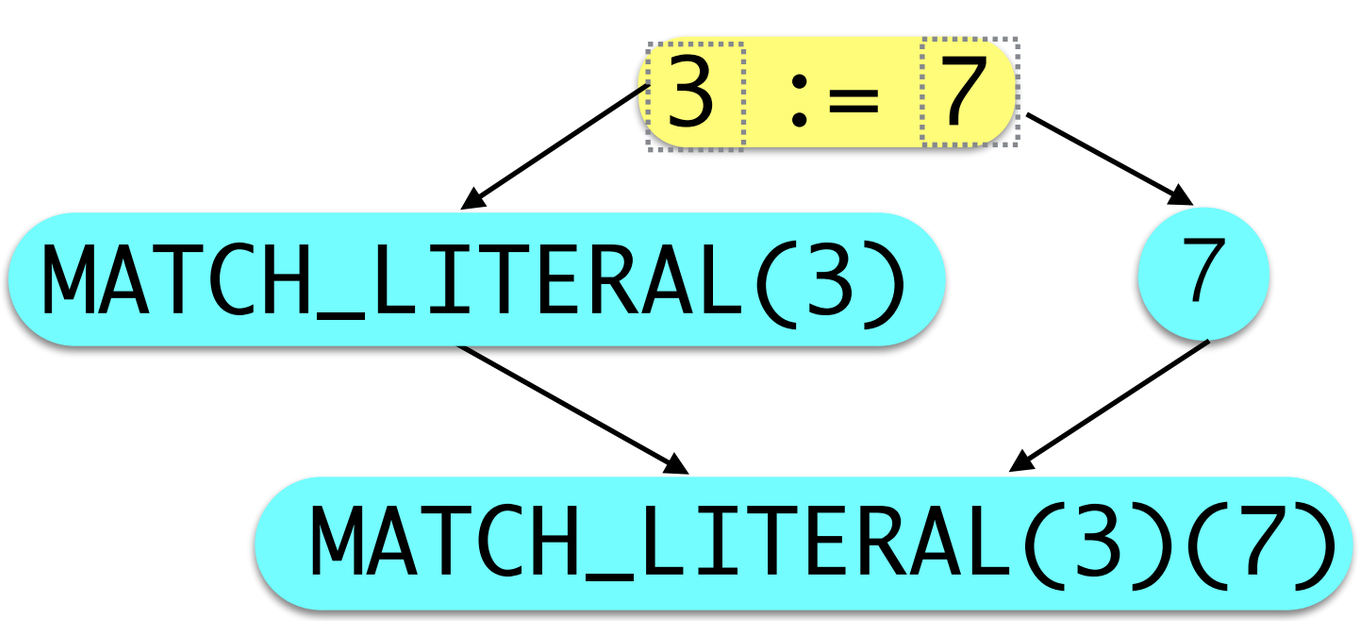
- This is exactly how list pattern matching works in Rascal.
- Matching of sets and maps is similar.
- One can explore all solutions in for-loops and conditions, this implies **local backtracking**
- Rascal's pattern language is much richer and may contain: *nodes*, *abstract datatypes*, *regular expressions*, *type constraints*, *anti-patterns*, *deep match, …*

How can we compile a match?



Matching Literals

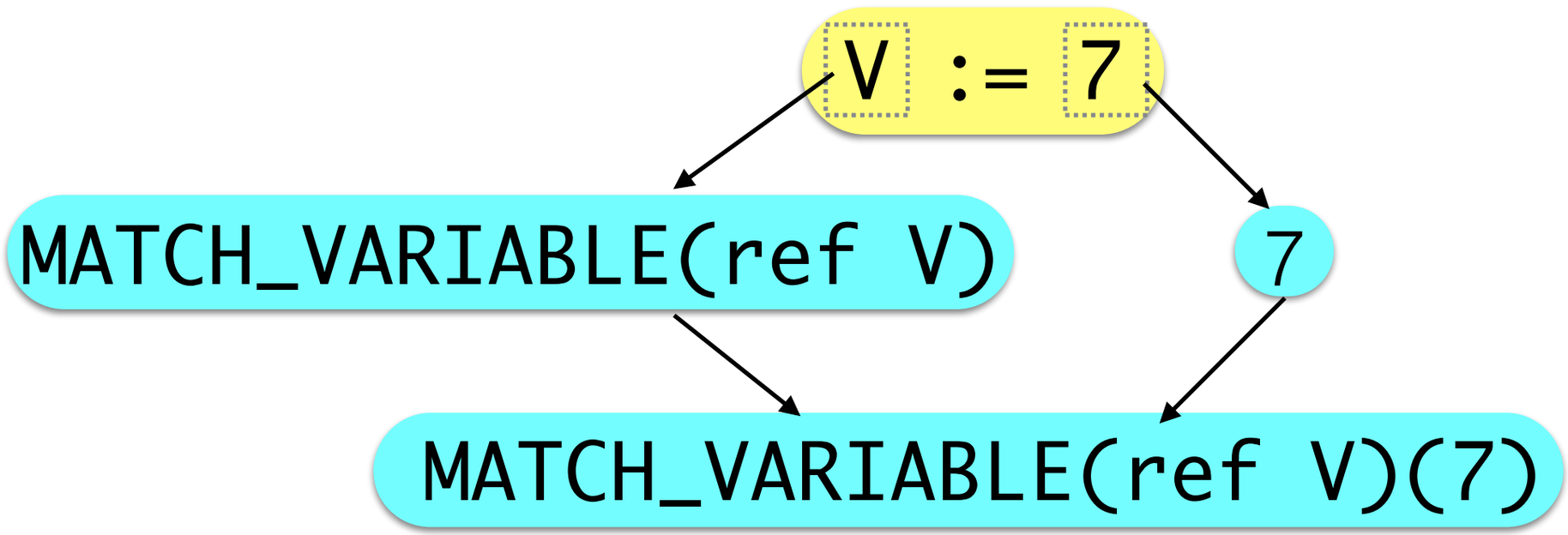
```
coroutine MATCH_LITERAL(pat, iSubject){
 if(equal(pat subject)) yield
}
```




```
if(next(create(MATCH_LITERAL(3)(7)))) …
```

Matching Variables

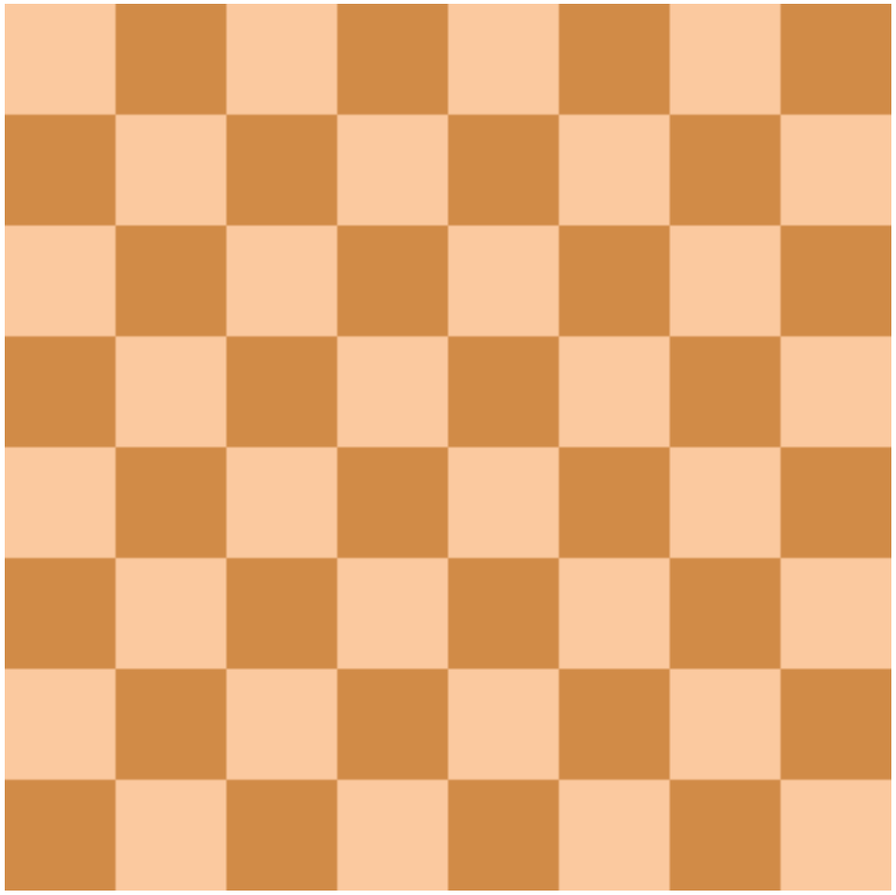
```
coroutine MATCH_VAR(pat, rVar, iSubject){
 deref rVar = iSubject; yield
}
```

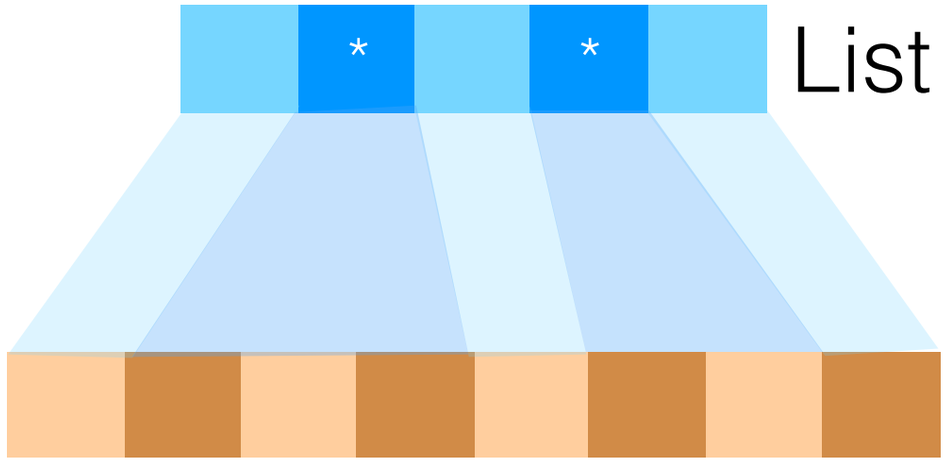



```
if(next(create(MATCH_VARIABLE(ref V)(7)))){…
    use V here …}
```

Match Lists

Recall the 8 queens problem





One of the purest examples of

generate-and-test backtracking

Match Lists

```
coroutine MATCH_LIST(pats, iSubject){
```
match pats from left to right:

On success:

if at end of subject:

yield a solution

else:

try to match pattern on the right

On failure:

if at begin of list:

fail

else:

try to match pattern on the left

}

```
coroutine MATCH_COLLECTION(pats,        // Coroutines to match collection elements
                           accept,      // Function that accepts a complete match
                           subject      // The subject
                           ) {
    var patlen = size_array(pats),      // Length of pattern array
        j = 0,                          // Cursor in patterns
        matchers                        // Currently active pattern matchers

    if(patlen == 0) {
        if(accept(subject)) {
            yield
        }
        exhaust
    }

    matchers = make_array(patlen)
    matchers[j] = create(pats[j], ref subject)
    while(true) {
        while(next(matchers[j])) {      // Move forward
            if((j == patlen - 1) && accept(subject)) {
                yield
            } else {
                if(j < patlen - 1) {
                    j = j + 1
                    matchers[j] = create(pats[j], ref subject)
                }
            }
        }
        if(j > 0) {                     // If possible, move backward
            j  = j - 1
        } else {
            exhaust
        }
    }
}
```

This code does not make many assumptions on the type of the subject!

Matching Lists

- Generating coroutine instances for lists and list variables is similar to (but slightly more complex than) the cases for literals and ordinary variables

- Details not shown today

Easily extensible

- This same algorithm
  - Can be instantiated to include length checks to speed up search
  - Can be instantiated for **set, bag** or **map** matching
- Only required:
  - subject data type
  - accept function on that subject

Effect of using coroutines to implement pattern matching

- Drastic reduction in implementation size (3.5-17x)
- Dramatic increase in understandability
  - **Full disclosure**: nobody understands the original Java code for list/set matching anymore (including the authors)
- Increase in speed (but hard to measure in isolation, observed 0.8-4x)

We use this technique in the Rascal compiler

Feature	Interpreter Java	Compiler muRascal	Reduction
Literal pattern	88	5	**17.6x**
List pattern	647	150	**4.3x**
Set pattern	791	223	**3.5x**
Node pattern	485	29	**16.7x**
Anti pattern	75	8	**9.3x**

Status Rascal Compiler

Current

- The Rascal compiler translates full Rascal to muRascal.
  - It already passes all 3400 tests of the Rascal test suite
  - Preparing to bootstrap

Planned

- Introduction of stackful coroutines in Rascal itself

- Coroutines enable good abstraction of data+control
- Problem areas with complex data+control can profit:
  - search, producer/consumer, event loops, GUIs, …
- Coroutines bridge gap between eager and lazy evaluation
- Despite the lack in standardization of terminology, coroutines are a worthwhile concept to consider
- Case in point: list/set/map matching in the Rascal compiler

Software Engineers and computer scientists lack historical perspective

=

they reinvent the wheel but do not improve it

Coroutines unify iterators, generators, streams, fibers, reactive programming, ...

- http://www.rascal-mpl.org
- http://tutor.rascal-mpl.org
- http://stackoverflow.com/questions/tagged/rascal