Control Flow

The way we move through the data that is the program,
doing things on our way

First, let's take a look at history.

This is a program that has been written
(probably between 1940 and 1970)
on an IBM 444
(a programmable accounting calculator)

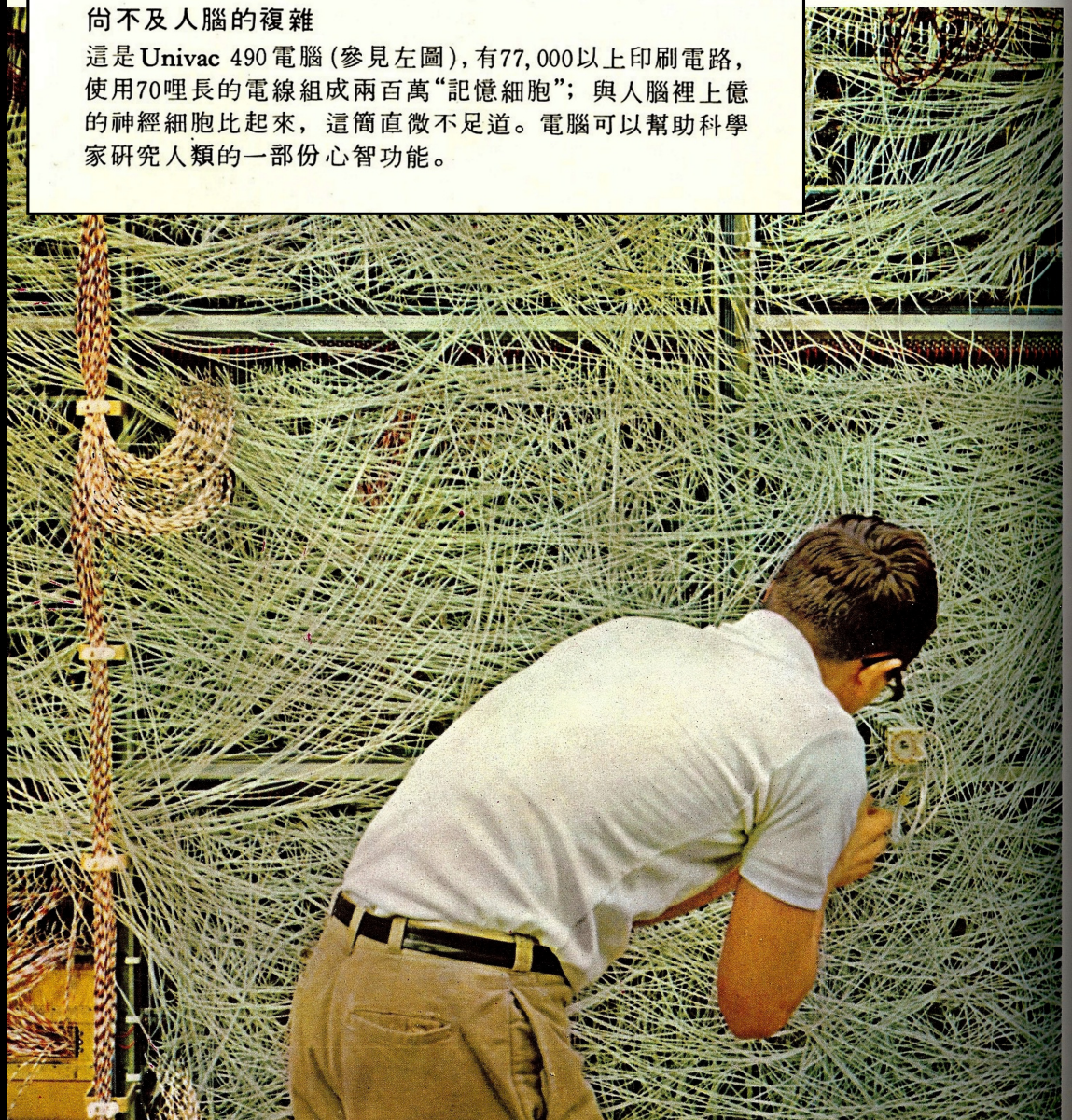
And this is a scientist in the late 1950s / early 1960s
doing neurological / psychological research

8

心理與人類的未來

尚不及人腦的複雜

這是Univac 490電腦(參見左圖),有77,000以上印刷電路,使用70哩長的電線組成兩百萬"記憶細胞";與人腦裡上億的神經細胞比起來,這簡直微不足道。電腦可以幫助科學家研究人類的一部份心智功能。



This is what a program library looks like
when you're using plugboards
as this Texan company still does in 2013

John von Neumann asserted in 1945 that programs themselves can be represented as data.

The first computer to implement his ideas became operational in 1949.

It became possible to program by entering long series of numbers that represented machine code.

Then, Grace Hopper (1906–1992) had the idea to collect such programs in ‘libraries’ and to ‘call functions’ like you ‘call a book’ in a real-world library.

This led her to conceiving A-0, the world’s first compiler, which turned into commercial products ARITH-MATIC and MATH-MATIC.

Next, she came up with the idea to write programs in languages similar to English (and other natural languages!)—which we’re still doing today.

The first eloquent programming language in history was developed by Hopper between 1953 and 1959; she called it FLOW-MATIC; this would later evolve into COBOL.

Here is a FLOW-MATIC code snippet.

```
# sample.flow-matic
# A sample (slightly edited):

     ..........................................................................
 (0)  input  inventory  file-a price        file-b;
      output priced-inv file-c unpriced-inv file-d;
      hsp d.

     ..........................................................................
 (1)  compare product-no (a) with product-no (b);
      if greater  go to operation 10;
      if equal    go to operation 5;
      otherwise   go to operation 2.

     ..........................................................................
 (2)  transfer a to d.
 (3)  write-item d.
 (4)  jump to operation 8.

     ..........................................................................
 (5)  transfer a to c.
 (6)  move unit-price (b) to unit-price (c).
 (7)  write-item c.
 (8)  read-item a;
      if end of data go to operation 14.
 (9)  jump to operation 1.

     ..........................................................................
(10)  read item b;
      if end of data go to operation 12.
(11)  jump to operation 1.

     ..........................................................................
(12)  set operation 9 to go to operation 2.  # Holy Cow! Metaprogramming!
(13)  jump to operation 2.

     ..........................................................................
(14)  test product-no (b) against zzzzzzzzzzz;
```

one can immediately see the huge step forward
that Hopper had made

code execution happens from top to bottom
with explicitly labelled GOTOs

...which is conceptually simple, but doesn't scale well
already in 1968, Dijkstra wrote his famous
GOTO considered harmful

‘simple’ doesn’t always equal ‘readable’

when branching, most of the time the condition comes before the conclusion:

```
if a < 0 then print "negative"
```

but sometimes we seem to be happier with the inversion of that (postfix if, possibly introduced by perl):

```
print "negative" if a < 0
```

FORTH has maybe the most ‘logical’ control flow

every line is strictly read from left to right

```
: FLOOR5 DUP 6 < IF DROP 5 ELSE 1 - THEN ;
```

everything is either a data item or a postfix operator

observe however it was felt necessary to make word definition a bracketing operator

who uses a stack-based language these days?

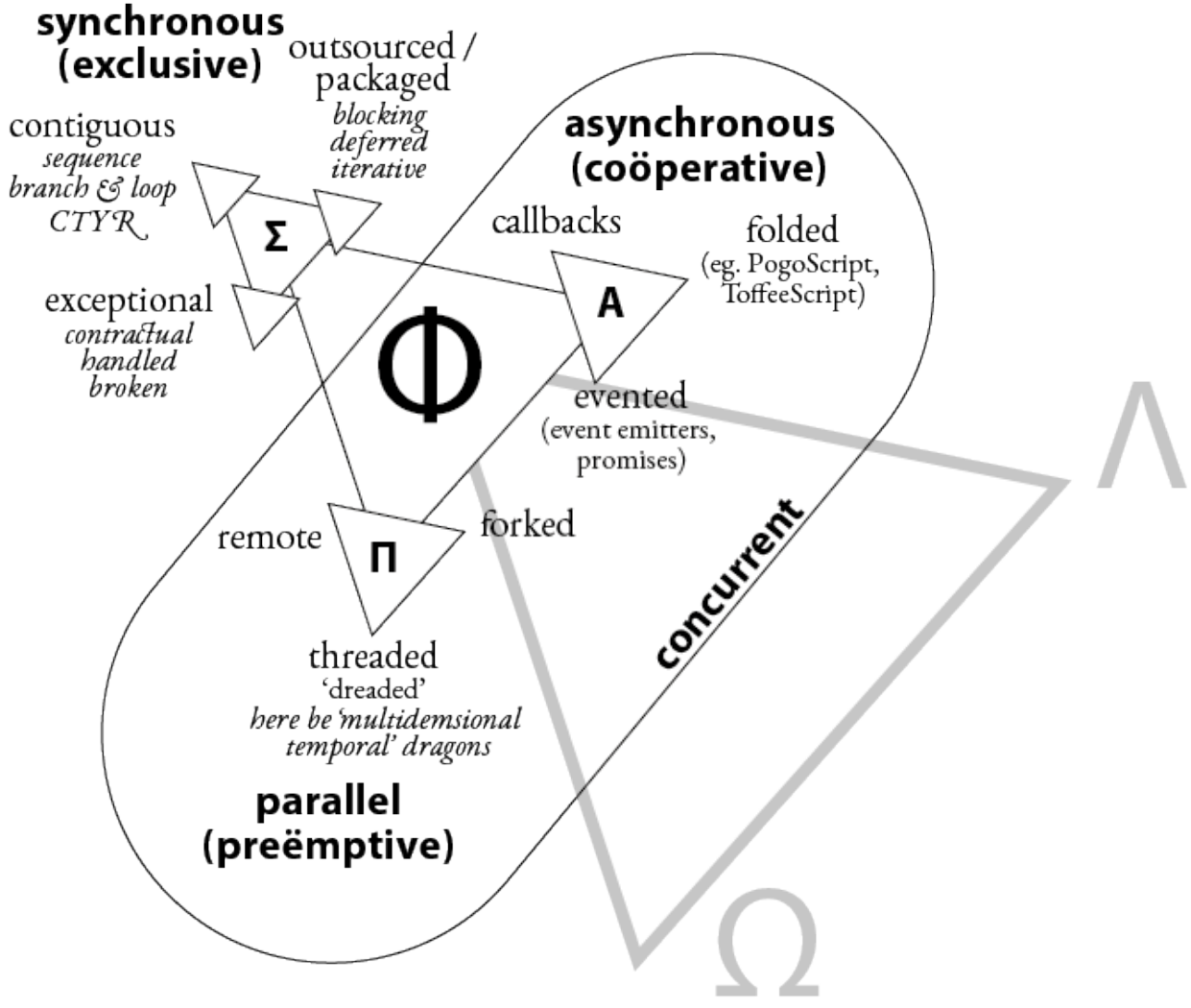
LISP is not all too popular, either

in FLOW-MATIC,
subroutine calls could be either called
in their original location on tape (incurring a time penalty)
or be inlined (incurring a space penalty)

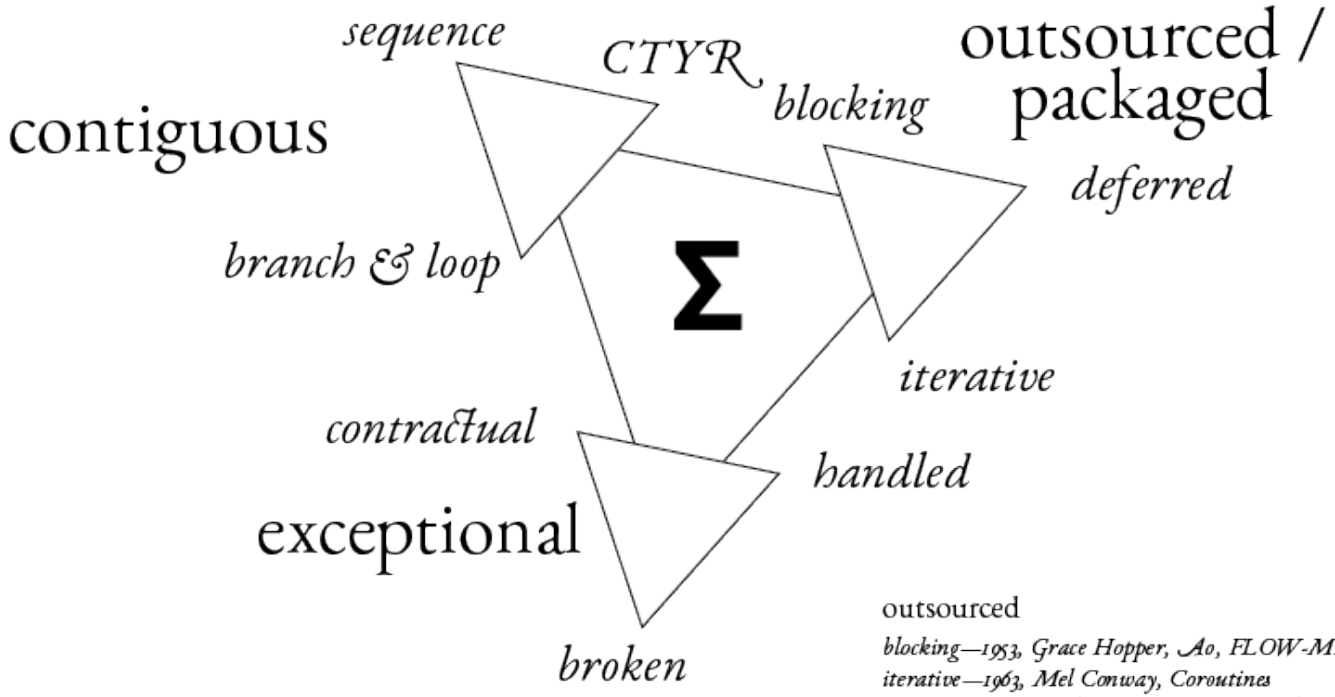
this is the origin of functions and macros

(TeX comes to mind, a programming language that
does everything the inline/expanding way)

today, the options for control flow have multiplied


synchronous
(exclusive)
outsourced /
packaged
blocking
deferred
iterative
contiguous
sequence
branch & loop
CTYR
Σ
exceptional
contractual
handled
broken
asynchronous
(coöperative)
callbacks
folded
(eg. PogoScript,
ToffeeScript)
A
Φ
evented
(event emitters,
promises)
remote
Π
forked
concurrent
threaded
'dreaded'
here be 'multidemsional
temporal' dragons
parallel
(preëmptive)
Λ
Ω

synchronous (exclusive)

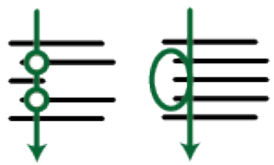


outsourced

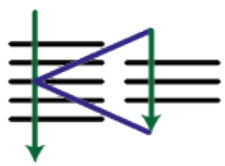
blocking—1953, Grace Hopper, A0, FLOW-MATIC
iterative—1963, Mel Conway, Coroutines
deferred—1973, Carl Hewitt, Actor Model

contiguous

sequence



branch & loop



CTYR (call; throw, yield, return)

contiguous

CTYR (`call`, `throw`, `yield`, `return`)
are basic constructs that break out of contiguous mode

like subtraction and division break out of
natural-number and integer-number mode

`call` may be non-blocking, which leads to asynchronous mode;
`throw` gives you exceptional mode;
`yield` gives you iteration;
`return` goes back to … anywhere! Kansas maybe!

exceptional

contractual

each language function(ality) has a usage contract
when that contract gets violated, an ‘exception is thrown’
execution recursively ‘falls back’ to caller
exception may be ‘handled’ with a `try` / `catch` clause

exceptions contract

(1) exceptions are thrown ifff there is a breach of contract

(2) exceptions carry a ‘routing slip’ (a stacktrace)
with all the file names, function names, and line numbers
they fell through

(3) you can catch them within a `catch` clause

exceptional

contractual, handled

example:

```
# try-catch-stacktrace.coffee

f = ->
  try
    Rg = g()
  catch Rx              # note: imagine `Rx = catch` or `catch as Rx` here
    ...

g = -> return h()
h = -> throw Rh
```

exceptional

contractual

`f` and `h` don't know about each other
but if `f` catches an exception, it may have come from `h`
so while `f` just wanted to talk to `g`
suddenly it's being talked to by `h`
this can be hard to grasp

exceptional

handled

control flow in Python's exception handling constructs *can* be difficult:

```
try:
  do_1a()
  do_1b()                                  # only executed if `do_1a` was OK

except Exception_class_2a as error_2a:     # only up to one `except` clause gets executed
  do_2a()

except Exception_class_2b as error_2b:
  do_2b()

else:                                      # only if `do_1x` was OK; used to avoid catching exceptions
  do_3()                                   # from other sources than `do_1x`

finally:                                   # executed after `do_3`; exceptions will be re-raised
  do_4()
```

(originally about why i never use Python's `for` / `in` / `else`)
Nr. 1 when googling "python else clause in for in statement":

conceptual stackoverflow.com

—I've noticed the following code is legal in Python. My question is why? Is there a specific reason?

```
n = 5
while n != 0:
    print n
    n -= 1
else:
    print "what the..."
```

—Wow, I never even knew about this.

—@detly: That's because most people avoid this construct. :) I believe Guido mentioned during the Py3k process that, at the very least, the choice of the word `else` for this use had been a remarkably bad idea, and that they wouldn't be doing any more of these.

exceptional

broken

example:

```
# try-catch-no-stacktrace.coffee
f = -> f()
f()
```

in JavaScript the exceptions contract is broken here
because of missing stacktrace
which can be very annoying to debug

exceptional

broken

Java and some frameworks are dreaded for their long stacktraces

some exceptions in JavaScript do not have stacktraces at all

exceedingly long stacktraces
missing stacktraces
incomplete stacktraces (resulting from asynchronous calls)
falsely positive and negative exceptions
and exceptions you can't catch in a `catch` clause
may be regarded as **broken exceptions**

exceptional

contractual, handled (and slightly broken)

example:

```
# try-catch-0.coffee

f = -> log g()
g = -> return h()
h = -> return k()
k = -> return d[ 'x' ]
f()
```

exceptions as a tool

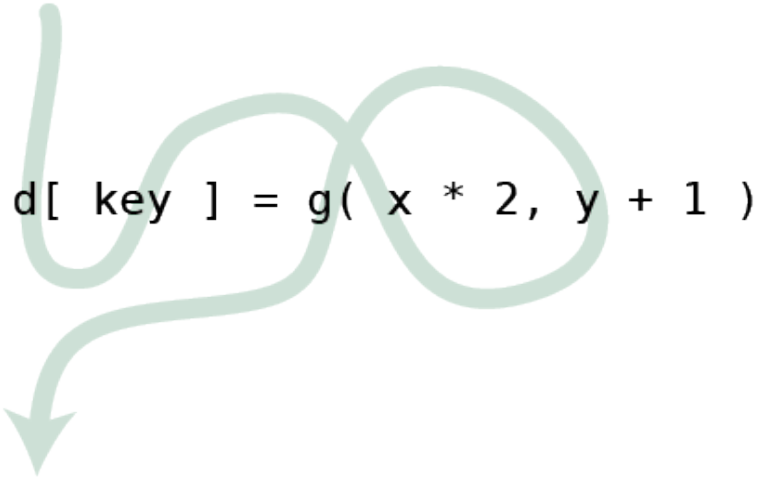
inline expression control flow—can't be so difficult!!?

```
d[ key ] = g( x * 2, y + 1 )
```

Q: in what order are the sub-expressions evaluated?

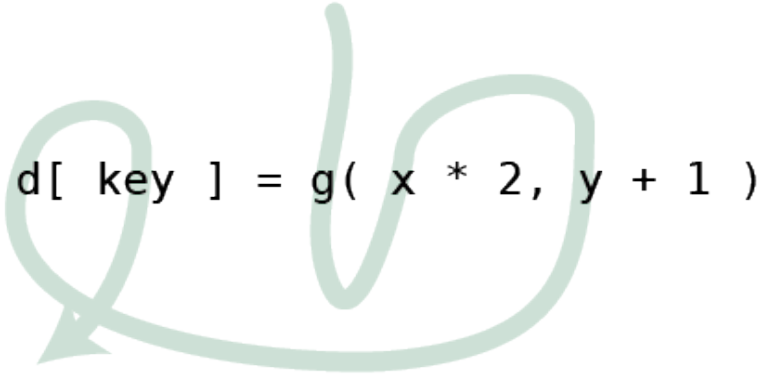
A: it depends!

```
# order-of-evaluation
```




```
d[ key ] = g( x * 2, y + 1 )

d
    key
                x * 2
                        y + 1
            g
```




```
d[ key ] = g( x * 2, y + 1 )

            g
                x * 2
                        y + 1
d
    key
```

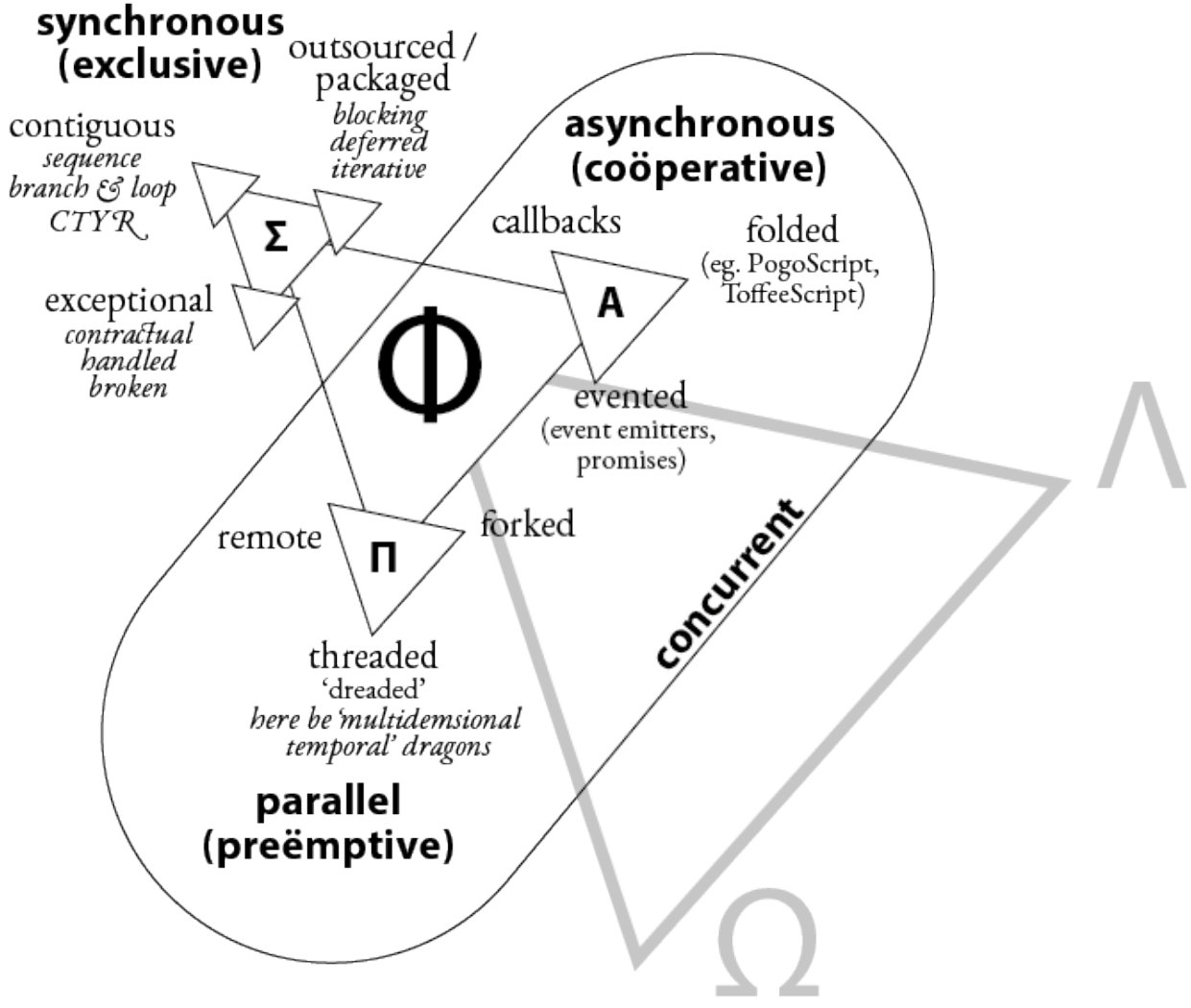
clearly, it would be nice to have a good way to picture control flow

but all that exists is of marginal popularity

(UML ‘Activity Diagrams’ are … behold … Flowcharts!)

is visualized control flow doomed like 3D desktops
& graphical programming languages?

... back to the beginning


synchronous
(exclusive)
outsourced /
packaged
blocking
deferred
iterative
contiguous
sequence
branch & loop
CTYR
Σ
exceptional
contractual
handled
broken
asynchronous
(coöperative)
callbacks
folded
(eg. PogoScript,
ToffeeScript)
A
Φ
evented
(event emitters,
promises)
remote
Π
forked
concurrent
threaded
'dreaded'
here be 'multidemsional
temporal' dragons
parallel
(preëmptive)
Λ
Ω

outsourced / packaged

blocking

code in function runs to completion
it is completed when `return` is encountered

iterative

code runs until `yield` is encountered
`yield` can return a value
function may resume; state is preserved in closure

deferred

a function `f` is called that returns ‘immediately’
it may cause an action `g` to run in parallel
an event `e` may become available at the earliest on the next turn

parallel / preëmptive

forked

code runs in other process on same machine

remote

code runs on another CPU

threaded / dreaded

several instances of the same process run in the same shared memory

asynchronous / coöperative

http://ejohn.org/blog/how-javascript-timers-work

```
# intervals.coffee
```

callbacks

pass in callback function when calling an async method
return value of async method most of the time throwaway value
most of NodeJS API works like this

evented

get back Event Emitter on calling async method
can attach handlers to events on this emitter
JS Promises/A+ builds on this

folded

code looks like it's synchronous—but it's not!

asynchronous / coöperative
... and ... handling exceptions ...?

exceptions thrown by asynchronous code
can not be handled in a `try` / `catch` clause

in NodeJS, you can use `process.on 'uncaughtException'`
and that's what i do in coffeenode-stacktrace:

```
# This is *so* 1990s VBA!
process.on 'uncaughtException', ( error ) =>
  @log_stacktrace error
```

asynchronous folded code

discovered this is @ http://pogoscript.org
& https://github.com/jiangmiao/toffee-script:

```
read_file '/tmp/whatever.txt', ( error, text ) ->
  throw error if error?
  log "here is the text:", text
log "the file is being read; results to be announced shortly"
```

becomes:

```
log "going to read file"
text = read_file! '/tmp/whatever.txt'
log "here is the text:", text
```

or even:

```
log "going to read file"
try
  text = read_file! '/tmp/whatever.txt'
catch error
  throw error
log text
```

asynchronous folded code

discovered this is @ http://pogoscript.org
& https://github.com/jiangmiao/toffee-script:

```
read_file '/tmp/whatever.txt', ( error, text ) ->
  throw error if error?
  log "here is the text:", text
log "the file is being read; results to be announced shortly"
```

becomes:

```
log "going to read file"
text = read_file! '/tmp/whatever.txt'
log "here is the text:", text
```

or even:

```
log "going to read file"
try
  text = read_file! '/tmp/whatever.txt'
catch error
  throw error
log text
```

asynchronous folded code

observations:

it does look tempting!

but what if you wanted to have code executed right after the call??

how do they do it?

well, maybe that's part of the problem...

Lo and Behold.
Just look at it.

```
# asynchronous-folded-pogo.pogo
# asynchronous-folded-pogo.js

# async-try-catch.pogo
# async-try-catch.js
```

kthxbye

https://github.com/loveencounterflow/CONTROLFLOW