STRUCTURED CONCURRENCY AND FLOW BASED PROGRAMMING

KERIPY ASYNC ARCHITECTURE - HIO

WHY USE STRUCTURED CONCURRENCY?

GOTO STATEMENT CONSIDERED HARMFUL

Edgar Dijkstra

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of $(A1, A2, \cdots, An)$), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1, B2 \rightarrow E2, \cdots, Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

REFERENCES:

1. WIRTH, NIKLAUS, AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. BÖHM, CORRADO, AND JACOPINI, GUISEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9* (May 1966), 366–371.

EDSGER W. DIJKSTRA
Technological University
Eindhoven, The Netherlands

https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually,

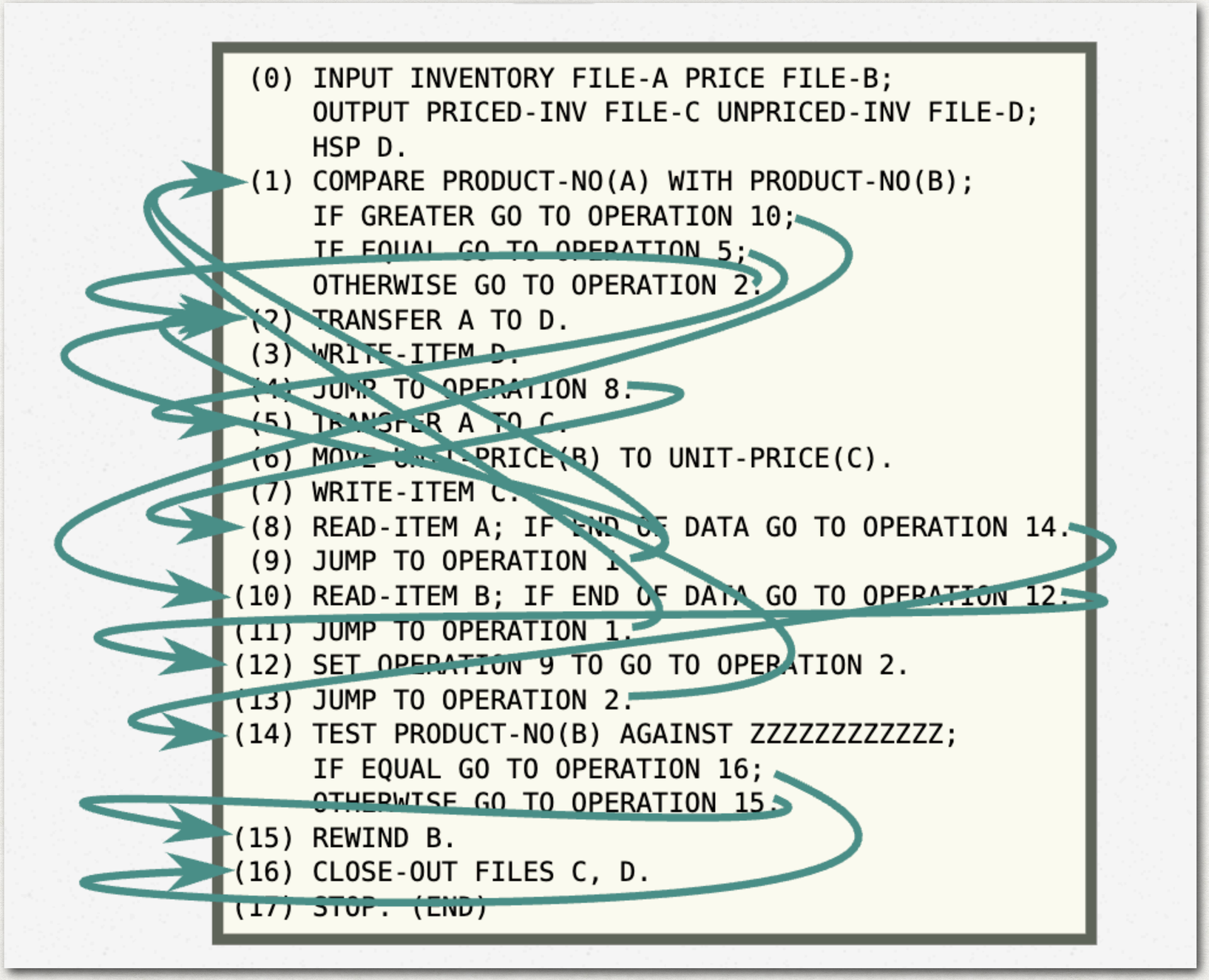
GOTO, GO, PROMISES,
CALLBACKS
BREAK ABSTRACTIONS
AND AUTOMATIC RESOURCE
CLEANUP

NOTES ON STRUCTURED CONCURRENCY; OR: GO TO STATEMENT CONSIDERED HARMFUL

Nathaniel J. Smith

https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/

```
 (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.
(13) JUMP TO OPERATION 2.
(14) TEST PRODUCT-NO(B) AGAINST ZZZZZZZZZZZZ;
     IF EQUAL GO TO OPERATION 16;
     OTHERWISE GO TO OPERATION 15.
(15) REWIND B.
(16) CLOSE-OUT FILES C, D.
(17) STOP. (END)
```


(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.
(13) JUMP TO OPERATION 2.
(14) TEST PRODUCT-NO(B) AGAINST ZZZZZZZZZZZZ;
IF EQUAL GO TO OPERATION 16;
OTHERWISE GO TO OPERATION 15.
(15) REWIND B.
(16) CLOSE-OUT FILES C, D.
(17) STOP. (END)

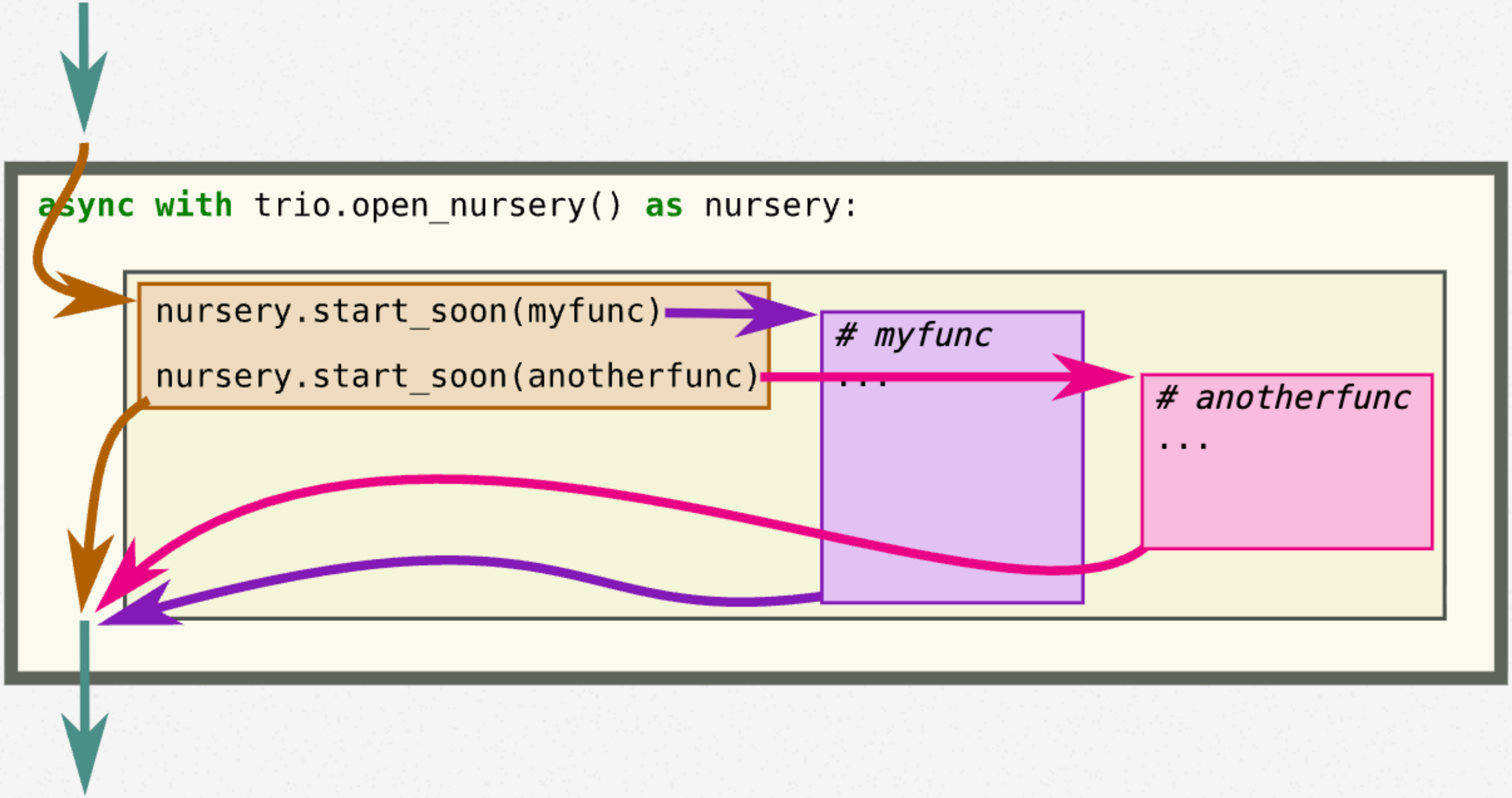
WHAT IS STRUCTURED CONCURRENCY

STRUCTURED CONCURRENCY

HIGH POINTS

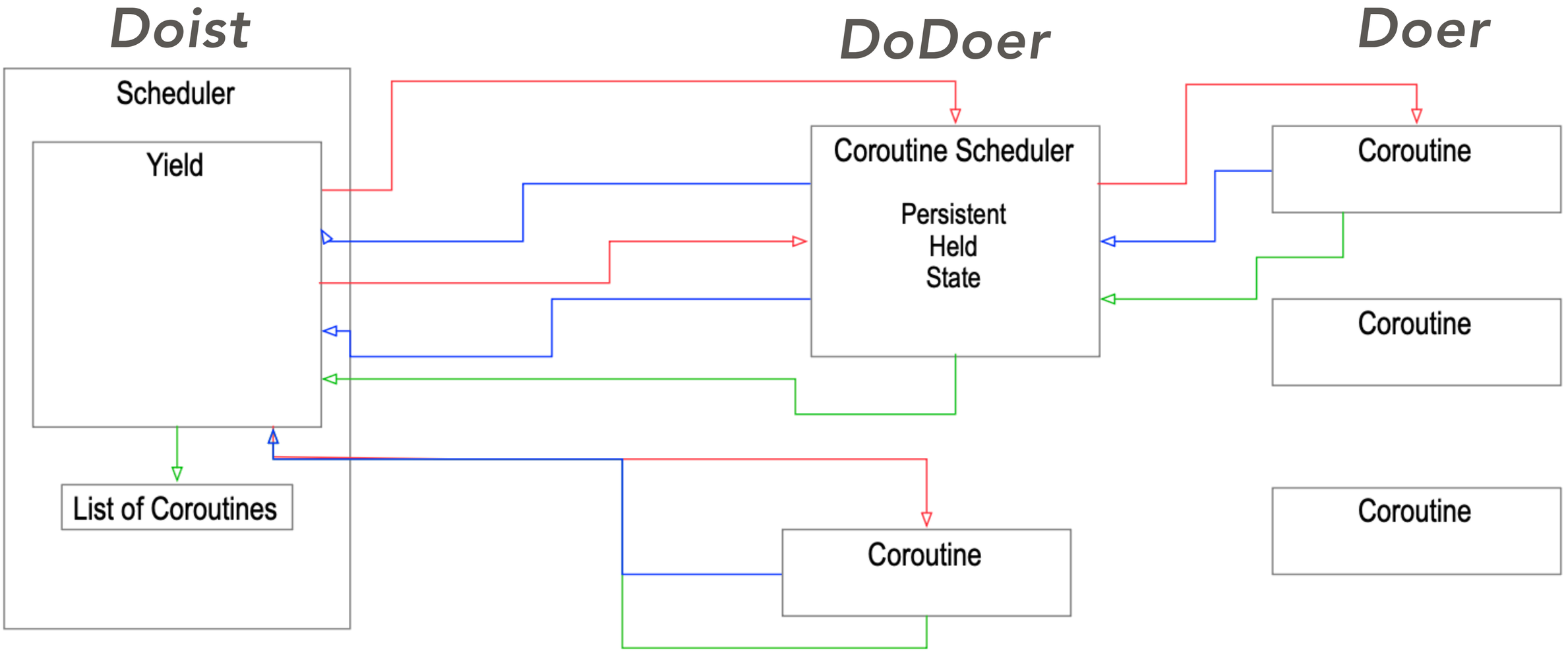
- Program (Task) Flow follows Lexical Scope
  - Avoids Promises, async/await, callbacks - avoids GOTO constructs
- Child tasks cannot outlive their parent scope
- Spawn implies join (DoDoer owns child Doers, Doist owns DoDoer/Doer children)
- Failure and cancellation propagate through the task tree
  - Exceptions are carried all the way up the scheduling tree to the Doist
    - Consistent, predictable exception behavior

TRIO AND NURSERIES


async with trio.open_nursery() as nursery:
nursery.start_soon(myfunc)
nursery.start_soon(anotherfunc)
# myfunc
...
anotherfunc
...

HIO AND DOERS

Hierarchical Coroutine



HIO INTERNALS

HIO INTERNALS

HOW KERIPY USES HIO

- Root Scheduler for Coroutines
  - Doist
- Coroutine Tasks - “Co-operative” coroutines
  - Doers
  - DoDoers
- Time in Async Systems
  - Tock - Tymist, et. al.

GETTING STARTED

LET'S FOLLOW PROGRAM FLOW

- ```
  kli incept \
     --name test1 \
     --alias test1 \
     --file ./my-incept-config.json
  ```
  ```

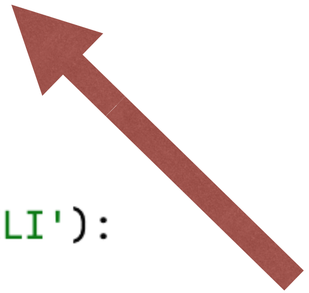
kli.py

```
def main():  Philip Feairheller +3
    parser = multicommand.create_parser(commands)
    args = parser.parse_args()

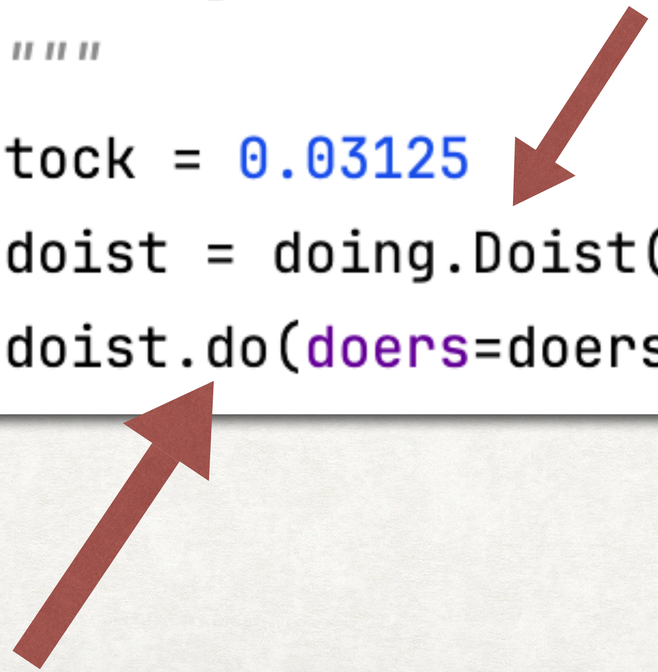
    if not hasattr(args, 'handler'):
        parser.print_help()
        return

    try:
        doers = args.handler(args)
        directing.runController(doers=doers, expire=0.0)

    except Exception as ex:
        import os
        if os.getenv('DEBUG_KLI'):
            import traceback
            traceback.print_exc()
        else:
            print(f"ERR: {ex}")
        return -1
```



```
def runController(doers, expire=0.0):  27 usages  Samuel M Smith *
    """
    Utility Function to create doist to run doers
    """
    tock = 0.03125
    doist = doing.Doist(limit=expire, tock=tock, real=True)
    doist.do(doers=doers)
```



DOIST

ROOT SCHEDULER

```
class Doist(tyming.Tymist):  Samuel M Smith +1
    """..."""
    def __init__(self, *, name='doist', real=False, limit=None, doers=None,
                         temp=False, **kwa):...


    def do(self, doers=None, limit=None, tyme=None, *, temp=None):...


    def enter(self, doers=None, *, temp=None):...


    def recur(self, deeds=None):...

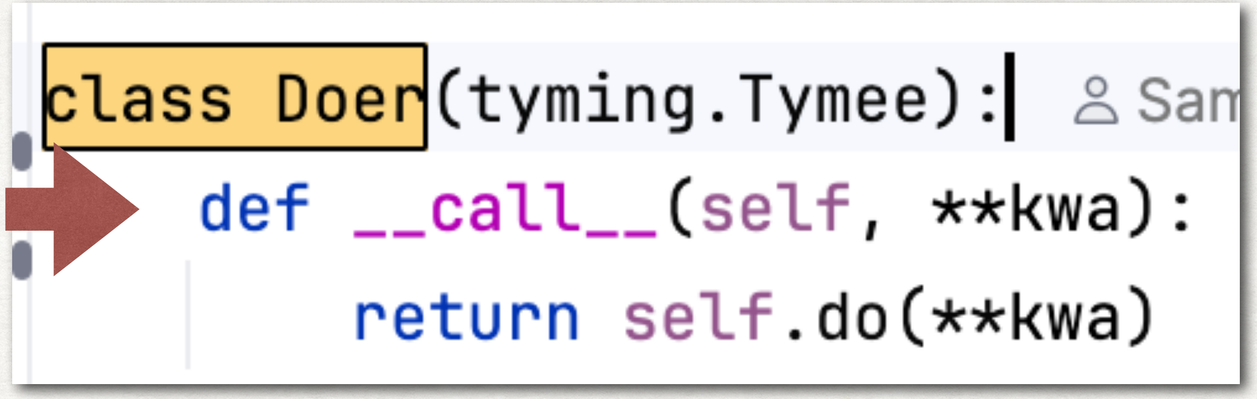

    def exit(self, deeds=None):...


    def extend(self, doers):...


    def remove(self, doers):...
```

```python
class Doist(tyming.Tymist):  Samuel M Smith +1 *
    ...
    def do(self, doers=None, limit=None, tyme=None, *, temp=None):  Samuel M Smith +1 *
        self.done = False
        if doers is not None:...
        if limit is not None:...
        if tyme is not None:...
        try:  # always clean up resources upon exception
            self.enter(temp=temp)  # runs enter context on each doer
            tymer = tyming.Tymer(tymth=self.tymen(), duration=self.limit)
            self.timer.start()
            while True:  # until doers complete or exception or keyboardInterrupt
                try:
                    self.recur()  # increments .tyme runs recur context
                    if self.real:  # wait for real time to expire
                        while not self.timer.expired:
                            time.sleep(max(0.0, self.timer.remaining))
                        self.timer.restart()  #  no time lost
                    if not self.deeds:  # no deeds
                        self.done = True
                        break  # break out of forever loop
                    if self.limit and tymer.expired:  # reached limit before all dee
                        break  # break out of forever loop
                except KeyboardInterrupt:  # Forced shutdown due to SIGINT, use CNTL
                    break
                except SystemExit: # Forced shutdown of process via sys.exit()
                    raise
                except Exception:  # Forced shutdown due to uncaught exception
                    raise
        finally: # finally clause always runs regardless of exception or not.
            self.exit()  # force close remaining deeds throws GeneratorExit
```

```python
class Doist(tyming.Tymist):  Samuel M Smith +1 *
    ...
    def recur(self, deeds=None):  Samuel M Smith +1 *
        if deeds is None:...
        deeds.append((None, None, None))  # append run through once marker
        while deeds: # do while uses explicit break to exit while
            dog, retyme, doer = deeds.popleft()  # pop it off
            if not dog:  # Marker detected so this run through once has completed
                break  # break loop at marker signifies once through
            if retyme <= self.tyme:  # run it now
                try:  # send tyme. yield tock, tock may change during sended run
                    tock = dog.send(self.tyme)  # yielded tock == 0.0 means re-run asap
                except StopIteration as ex:  # returned instead of yielded
                    try:  # assign done state non forced return
                        doer.done = ex.value if ex.value is not None else doer.done
                    except AttributeError:  # bount method generator
                        # write to doer.__func__.done read from doer.done
                        doer.__func__.done = ex.value if ex.value is not None else doer.done
                else:  # reappend for next pass
                    if not tock:  # tock is None or tock == 0.0 with empty yield tock == None
                        retyme = self.tyme + self.tock  # rerun at next recur
                    else:
                        retyme += tock  # cumulative retyme of doer tock
                    deeds.append((dog, retyme, doer))  # reappend for next run through
            else:  # not retyme yet
                deeds.append((dog, retyme, doer))  # reappend for next pass
        self.tick()  # advance .tyme by one doist .tock
```


class Doer(tyming.Tymee):
def __call__(self, **kwa):
return self.do(**kwa)

```
class Doer(tyming.Tymee):  Samuel M Smith +1 *
    ...
    def do(self, tymth, *, tock=0.0, temp=None, **opts):  Samuel M Smith +1 *
        try:
            # enter context
            self.wind(tymth)  # update tymist dependencies
            self.tock = tock  #  s t tock to parameter
            self.enter(temp=temp)
            if isgeneratorfunction(self.recur):  #  .recur is generator method
                self.done = yield from self.recur(tock=self.tock)  # recur context delegated
            else:  # .recur is standard method so iterate in while loop
                while (not self.done):  # recur context
                    tyme = (yield (self.tock))  # yields .tock then waits for next send
                    self.done = self.recur(tyme=tyme)  # False means recur again
        except GeneratorExit:  # close context, forced exit due to .close on generator
            self.cease()
        except Exception as ex:  # abort context, forced exit due to uncaught exception
            self.abort(ex=ex)
            raise
        else:  # clean context
            self.clean()
        finally:  # exit context, exit, unforced if normal exit of try, forced otherwise
            self.exit()
        return self.done  # Only returns done state if normal return or close not abort raise
```

```
def recur(self, tyme):  Samuel M Smith +1 *
    """Do 'recur' context actions. Override in subclass.
    Regular method that perform repetitive actions once per invocation.
    Assumes resource setup in .enter() and resource takedown in .exit()
    (see ReDoer below for example of .recur that is a generator method)

    Returns completion state of recurrence actions.
       True means done False means continue
    ...
    """
    return (False)
```

Example

```
class MigrateDoer(doing.Doer):  2 usages  pfeairheller +1 *

    def __init__(self, args):...

    def recur(self, tyme):  pfeairheller +1 *
        db = basing.Baser(...)
        try:
            db.reopen()
        except kering.DatabaseError:
            pass
        print(f"Migrating {self.args.name}...")
        db.migrate()
        print(f"Finished migrating {self.args.name}")
        return True
```

WHY SO FEW DOERS
WHY SO MANY DODOERS
IN KERIPY?

DOING.DOIFY

YIELD FROM

```python
class DoDoer(Doer):  Samuel M Smith +1 *
    ...

    def do(self, tymth, tock=0.0, doers=None, always=None, *, temp=None, **opts):  Samuel M Smit
        always = always if always is not None else self.always
        if doers is not None:...
        try:
            # enter context
            self.wind(tymth)  # change tymist dependencies
            self.tock = tock  #  set tock to parameter
            # tyme = self.tyme
            self.enter(temp=temp)  # doist.enter() equivalent
            #recur context  self.done set by super doist or dodoer prior to enter
            while (not self.done or always):  # recur context
                tyme = (yield (self.tock))  # yields .tock then waits for next send
                self.done = self.recur(tyme=tyme)  # equv of doist.recur
        except GeneratorExit:  # cease context, forced exit due to generator.close()
            self.cease()
        except Exception as ex:  # abort context, forced exit due to uncaught exception
            self.abort(ex=ex)
            raise
        else:  # clean context
            self.clean()
        finally:  # exit context, exit, unforced if normal exit of try, forced otherwise
            self.exit()  # equiv of doist.do finally clause
        return self.done  # Only returns done state if normal return not close or abort raise
```

lifecycle fn

lifecycle fn

lifecycle fn

lifecycle fn

lifecycle fn

lifecycle fn

```
class Doer(tyming.Tymee):  Samuel M Smith +1 *
    ...
    def do(self, tymth, *, tock=0.0, temp=None, **opts):  Samuel M Smith +1 *
        try:
            # enter context
            self.wind(tymth)  # update tymist dependencies
            self.tock = tock  #  set tock to parameter
            self.enter(temp=temp)
            if isgeneratorfunction(self.recur):  #  .recur is generator method
                self.done = yield from self.recur(tock=self.tock)  # recur context delegated
            else:  # .recur is standard method so iterate in while loop
                while (not self.done):  # recur context
                    tyme = (yield (self.tock))  # yields .tock then waits for next send
                    self.done = self.recur(tyme=tyme)  # False means recur again
        except GeneratorExit:  # close context, forced exit due to .close on generator
            self.cease()
        except Exception as ex:  # abort context, forced exit due to uncaught exception
            self.abort(ex=ex)
            raise
        else:  # clean context
            self.clean()
        finally:  # exit context, exit, unforced if normal exit of try, forced otherwise
            self.exit()
        return self.done  # Only returns done state if normal return or close not abort raise
```

Generator execution delegation

lifecycle fn

lifecycle fn

lifecycle fn

lifecycle fn

lifecycle fn

lifecycle fn

DOING.DOIFY

```python
class InceptDoer(doing.DoDoer):  2 usages  Philip Feairheller +2 *
    """ DoDoer for creating a new identifier prefix and Hab with an alias.
    """


    def __init__(self, name, base, alias, bran, endpoint, proxy=None, cnfg=None, **kwa):
        ...
        doers = [
            self.hbyDoer,
            self.postman,
            self.mbx,
            self.swain,
            doing.doify(self.inceptDo)
        ]

        self.inits = kwa
        self.alias = alias
        super(InceptDoer, self).__init__(doers=doers)
```



spawn

inline-lexical yield

inline-lexical yield

inline-lexical yield

inline-lexical yield

join-ish

```
class InceptDoer(doing.DoDoer):  1 usage  Philip Feairheller +2 *
    ...
    def inceptDo(self, tymth, tock=0.0, **kwa):  Philip Feairheller +2 *
        ...
        hab = self.hby.makeHab(name=self.alias, **self.inits)
        witDoer = agenting.WitnessReceiptor(hby=self.hby)
        receiptor = agenting.Receiptor(hby=self.hby)
        self.extend([witDoer, receiptor])
        if hab.kever.delpre:
            self.swain.delegation(pre=hab.pre, sn=0)
            print("Waiting for delegation approval...")
            while not self.swain.complete(hab.kever.prefixer, coring.Seqner(sn=hab.kever.sn)):
                yield self.tock
        elif hab.kever.wits:
            print("Waiting for witness receipts...")
            if self.endpoint:
                yield from receiptor.receipt(hab.pre, sn=0)
            else:
                witDoer.msgs.append(dict(pre=hab.pre))
                while not witDoer.cues:
                    _ = yield self.tock
        if hab.kever.delpre:
            if self.proxy is not None:
                sender = self.proxy
            else:
                sender = hab
            yield from self.postman.sendEventToDelegator(hab=hab, sender=sender, fn=hab.kever.sn)
        toRemove = [self.hbyDoer, witDoer, self.mbx, self.swain, self.postman, receiptor]
        self.remove(toRemove)
        return
```



Effection

Structured Concurrency and Effects for JavaScript

Get Started | API Reference

```ts
/**
 * Load identifiers through the connected runtime client and update Redux.
 */
export function* listIdentifiersOp(): EffectionOperation<IdentifierSummary[]> {  Show us
    const services : AppServices  = yield* AppServicesContext.expect();
    const identifiers : ({ name: string; prefix: string; icp_dt: ...  = yield* listIdentifiersService({
        client: services.runtime.requireConnectedClient(),
    });

    services.store.dispatch(
        identifierListLoaded({
            identifiers,
            loadedAt: new Date().toISOString(),
        })
    );

    return identifiers;
}
```

```ts
/**
 * Load and normalize every managed identifier visible to the connected client.
 */
export function* listIdentifiersService({  Show usages  Kent Bull *
    client,
}: {
    client: SignifyClient;
}): EffectionOperation<IdentifierSummary[]> {
    const response : any  = yield* callPromise(
      () : Promise<any>  => client.identifiers().list()
    );
    return identifiersFromResponse(response);
}
```

GOTCHAS

AVOID THESE WHEN WRITING WITH HIO

- Avoid generator priming causing business logic side effects
- Priming = advance to first yield
  This is a Python generator thing, not HIO specific.

```
# enter context
self.wind(tymth)
self.tock = tock
_ = (yield self.tock)
```

```
def inceptDo(self, tymth, tock=0.0, **kwa):  1 usage  Philip Feairheller +2 *
    # enter context
    self.wind(tymth)
    self.tock = tock
    _ = (yield self.tock)

    hab = self.hby.makeHab(name=self.alias, **self.inits)
    witDoer = agenting.WitnessReceiptor(hby=self.hby)
    receiptor = agenting.Receiptor(hby=self.hby)
    self.extend([witDoer, receiptor])

    if hab.kever.delpre:
        self.swain.delegation(pre=hab.pre, sn=0)
        print("Waiting for delegation approval...")
        while not self.swain.complete(hab.kever.prefixer, coring.Seqner(sn=hab.kever.sn)):
            yield self.tock
```

- The first **yield** is the barrier between **enter-time** setup and **recur-time** business logic.

AVOID OVERUSING DOIFY

OVERVIEW

ADVANCED - CORE CHARACTERISTICS

HOW HIO WITH KERIPY WORKS

- Cadence: tock chooses the next time a Doer will be serviced by a Doist
  - The next time Doer.recur() will run by Doist
- Queue: Deck / cues chooses next unit of work
- Transport: txbs / socket readiness determines whether bytes can leave

- Pressure flows upstream only when a slower downstream stage causes work to build up, or remain queued and upstream code respects that state

WHAT IS FLOW BASED PROGRAMMING

“

RESISTANCE OR FORCE OPPOSING THE DESIRED **FLOW OF DATA THROUGH SOFTWARE**.

— Jay Phelps (common back pressure definition)

”

NATURAL BACK PRESSURE

A CONSEQUENCE OF DETERMINISTIC SCHEDULING

- Natural back pressure at I/O boundaries
- Client.send is the heart of the mechanism:
  - queue bytes into txbs
  - attempt nonblocking send
  - if blocked, send 0
  - keep remainder in txbs
  - retry on later service cycle

- HIO does not implement a userland credit-based backpressure protocol in these HTTP/TCP classes.
- It implements buffered nonblocking I/O, and the real backpressure signal comes from the socket layer.

FLOW BASED PROGRAMMING IN KERIPY

FLOW BASED PROGRAMMING KERIPY

DOUBLE ENDED QUEUES FOR CROSS-TASK SIGNALING

- Double ended queues - decking.Deck
- Injected across components
- Some tasks produce
- Some tasks consume

ADAPTING WSGI TO ASYNC FOR STREAMING

`QryRpyMailboxIterable`

PYTHON LIBRARIES

Hio, Curio, Trio, asyncio

TYPESCRIPT LIBRARIES

Effection

END