Gordon Bell observed:

> *The cheapest, fastest and most reliable components of a computer system are those that aren't there.*

This has a parallel in data structures:

> *The fastest, most parsimonious, and best performing data structure is one which is never concretized. A promise to create data when–or if–it is needed is often easy to make.*

Gordon Bell observed:

> *The cheapest, fastest and most reliable components of a computer system are those that aren't there.*

This has a parallel in data structures:

> *The fastest, most parsimonious, and best performing data structure is one which is never concretized. A promise to create data when–or if–it is needed is often easy to make.*

The addition of iterators and generators to Python during the 2.x series, and their more systematic use in 3.x, provides an easy way to work with lazy computation.

Using these facilities well can improve program performance, often their big-O complexity.

Complex lazy data structures may require special design in order to encapsulate more complex promises than one can make with list-like iterators.

The addition of iterators and generators to Python during the 2.x series, and their more systematic use in 3.x, provides an easy way to work with lazy computation.

Using these facilities well can improve program performance, often their big-O complexity.

Complex lazy data structures may require special design in order to encapsulate more complex promises than one can make with list-like iterators.

The addition of iterators and generators to Python during the 2.x series, and their more systematic use in 3.x, provides an easy way to work with lazy computation.

Using these facilities well can improve program performance, often their big-O complexity.

Complex lazy data structures may require special design in order to encapsulate more complex promises than one can make with list-like iterators.

- Review of laziness
  - Examples from Functional Programming Languages
  - Generic *any language* laziness
- Iterators and the `itertools` module
- Generators and generator expressions
- Memoization and the `weakref` module
- Laziness in a directed acyclic graph
- Miscellaneous exoterica

(but not necessarily in the order listed)

# Laziness in a *really* lazy language (Haskell)

```
module Bounce where
  bounce :: Int -> Int
  bounce n = (n*379 + 522) `mod` 100000
  bseq :: Int -> [Int]
  bseq init = bounce init : map bounce (bseq init)
```

---

```
Bounce> take 8 (bseq 1)
[901,42001,18901,64001,56901,66001,14901,48001]
Bounce> bseq 1 !! 750
50901
```

(imagine **Bounce** as a really crude stand-in for, e.g. cipher-block chaining)

# Laziness by declaration of promises (Scheme)

```
> (define (bounce n) (modulo (+ (* n 379) 522) 100000))
> (define (bseq n) (let ((next (bounce n)))
                     (cons next (delay (bseq next)))))
> (display (car (bseq 1)))
901
> (display (cdr (bseq 1)))
#<promise:Bounce:4:53>
> (display (force (cdr (bseq 1))))
(42001 . #<promise:Bounce:4:53>)
```

# Iterators and Generators (remember 2001?)

- Iterators and generators are "sequence-like"
  - Potentially infinite length
  - Only need to concretize one element at a time
  - Hence cannot slice or index (*but wait a few slides*)
- An iterator is an object that has the methods `.next()` and `.__iter__()`. That's all!
- A generator is a powerful type of iterator: a *resumable function!*
  - Not quite a continuation, but more than a closure

# Iterators and Generators: An iterator example

```
class Iterator(object):
    def __init__(self, init=1, stop=None):
        self.n, self.stop = init, stop
    def next(self):
        if self.n == self.stop:
            raise StopIteration
        self.n = (self.n*379 + 522) % 100000
        return self.n
    def __iter__(self):
        return self
```

(remember that this is our crude stand-in for something expensive)

# Iterators and Generators: A generator example

```
def generator(init=1, stop=None):
    n = init
    while n != stop:
        n = (n*379 + 522) % 100000
        yield n
```

---

```
>>> for n in generator(): # for n in Iterator():
...     if not something_about(n):
...         break
...     do_stuff(n)           # return n w/ side effect
```

# Iterators and Generators: itertools module 1

```
>>> while n in generator():
...     if not something_about(n): break
...     do_stuff(n)          # return n w/ side effect
>>> from itertools import *
>>> ready = imap(do_stuff, takewhile(
...                   something_about, generator()))
>>> ready
<itertools.imap object at 0x19af890>
>>> list(ready)          # for n in ready: print n,
[901, 42001, 18901, 64001, 56901, 66001, 14901, 48001]
```

# Iterators and Generators: `itertools` module 2

```
>>> from itertools import *
>>> slice50_55 = islice(generator(), 50, 55)
>>> slice50_55
<itertools.islice object at 0x19a3ab0>
>>> list(slice50_55)
[50901, 92001, 68901, 14001, 6901]
>>> list(slice50_55)
[]
>>> g = generator(); list(islice(g, 3))
```

(what do we expect `g` to do if we keep `islice()`'ing it?)

# Iterators and Generators: generator expressions

```
# Cannot use listcomp on infinite generator
# E.g. [n**2 for n in generator() if n%3] blows up!
>>> not_div3 = (n**2 for n in generator() if n % 3)
>>> not_div3
<generator object <genexpr> at 0x21bab70>
>>> from itertools import *
>>> list(islice(not_div3, 3, 6))
[2704104001L, 2199703801L, 5776152001L]
```

# Things to avoid doing

(... *at this particular moment*):

- Expensive computations
- Concretize large data sets
- Time consuming background operations
    - Database queries
    - Retrieving network resources
    - Waiting for external events

(but the last one is the topic of some different presentation)

# A minimal class for delaying expensive actions 1

```
class Promise(object):
  def __init__(self, func, *args, **kws):
     self.func = func
     self.args = args
     self.kws = kws
  def __call__(self):
     if not hasattr(self, 'val'):
        self.val = self.func(*self.args,**self.kws)
     return self.val
```

# A minimal class for delaying expensive actions 2

```
>>> from promises import *
>>> p = Promise(slow_random)
>>> p
<promises.Promise object at 0x18ebb10>
>>> p.val
AttributeError: 'Promise' object has no attribute 'val'
>>> p()         # Eventually get the result
370754137
>>> p()         # Immediately get the result
370754137
```

# A slightly friendlier class for making promises

```
class Promise2(Promise):
    def forget(self):
        del self.val
    def __repr__(self):
        return repr(self())
    def __iter__(self):
        return iter(self())
    #...Some more magic methods could help too
```

(now we can concretize with **`print val`** or **`for x in val`**)

# Seamless promises inside data structures

```
class LazyDict(dict):
    def __getitem__(self, key):
        val = dict.__getitem__(self, key)
        if isinstance(val, Promise):
            val = val()
        return val
```

```
>>> ld = LazyDict(p=Promise(slow_random), n=99)
>>> print ld, ld['p']
{'p':<Promise object at 0x195f190>, 'n':99} 189636259
```

# Making promises forgetfully to save memory 1

```
import weakref
class WeakPromise(Promise):
    def __call__(self):
        if not hasattr(self, 'val'):
            val = self.func(*self.args, **self.kws)
            try: self.val = weakref.ref(val)
            except TypeError:
                self.val = val
        return self.val()
```

(notice `weakref` can only reference `object`, not `int`, `str`, etc.)

# Making promises forgetfully to save memory 2

```
>>> wp = WeakPromise(module.func, arg1, arg2)
>>> result = wp()
>>> print result
<module.SomeObj object at 0x1979670>
>>> print wp()
<module.SomeObj object at 0x1979670>
>>> del result
>>> print wp()
None
```

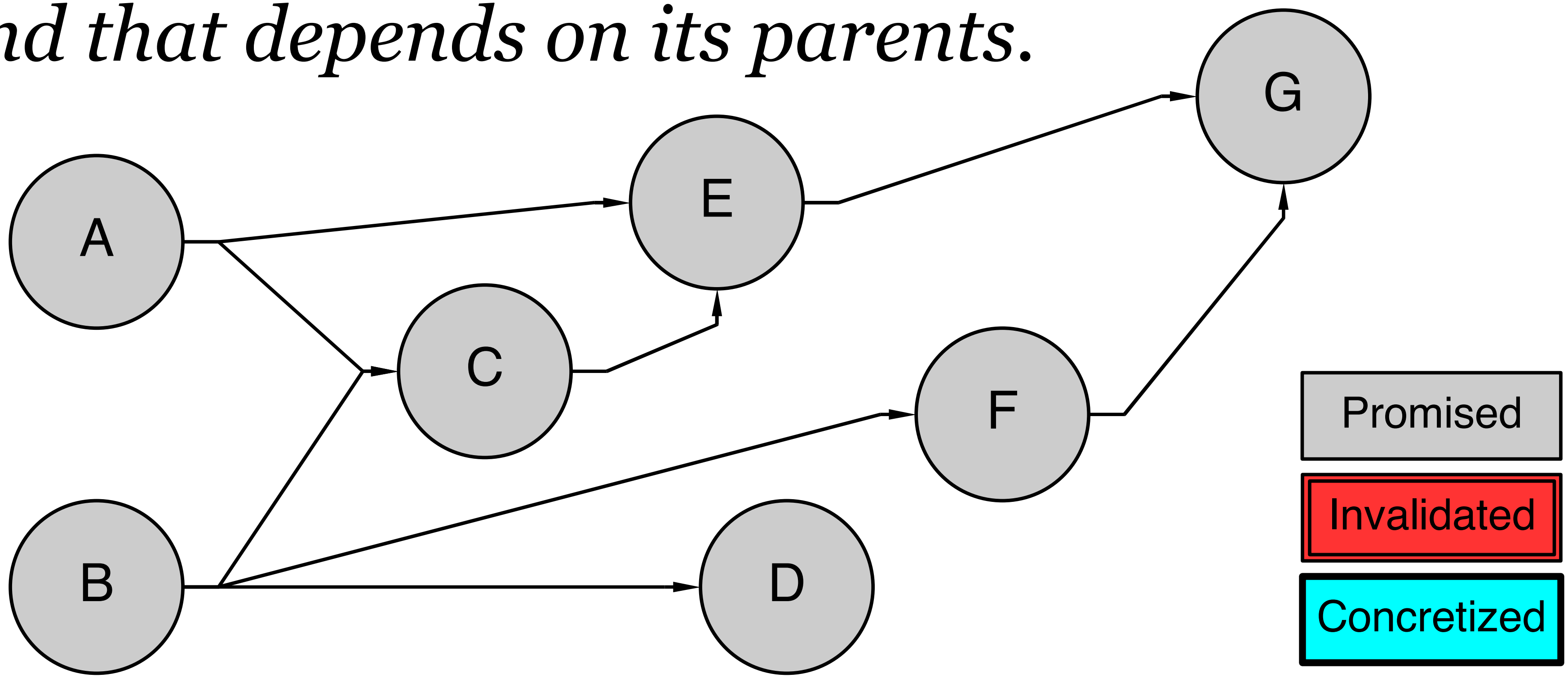
(if we want **`WeakPromise`** fulfilled anew, **`del wp.val`**)

# Trading memory for computation (memoization)

```
def memoize(fn):
    class Cached(object):
        def __init__(self, fn):
            self.fn, self.cache = fn, dict()
        def __call__(self, *args, **kws):
            key = (repr(args), repr(kws))
            self.cache[key] = self.cache.get(key) \
                              or self.fn(*args, **kws)
            return self.cache[key]
    return Cached(fn)
```

(the twin of a **`Promise`**; compute right away, but only once)

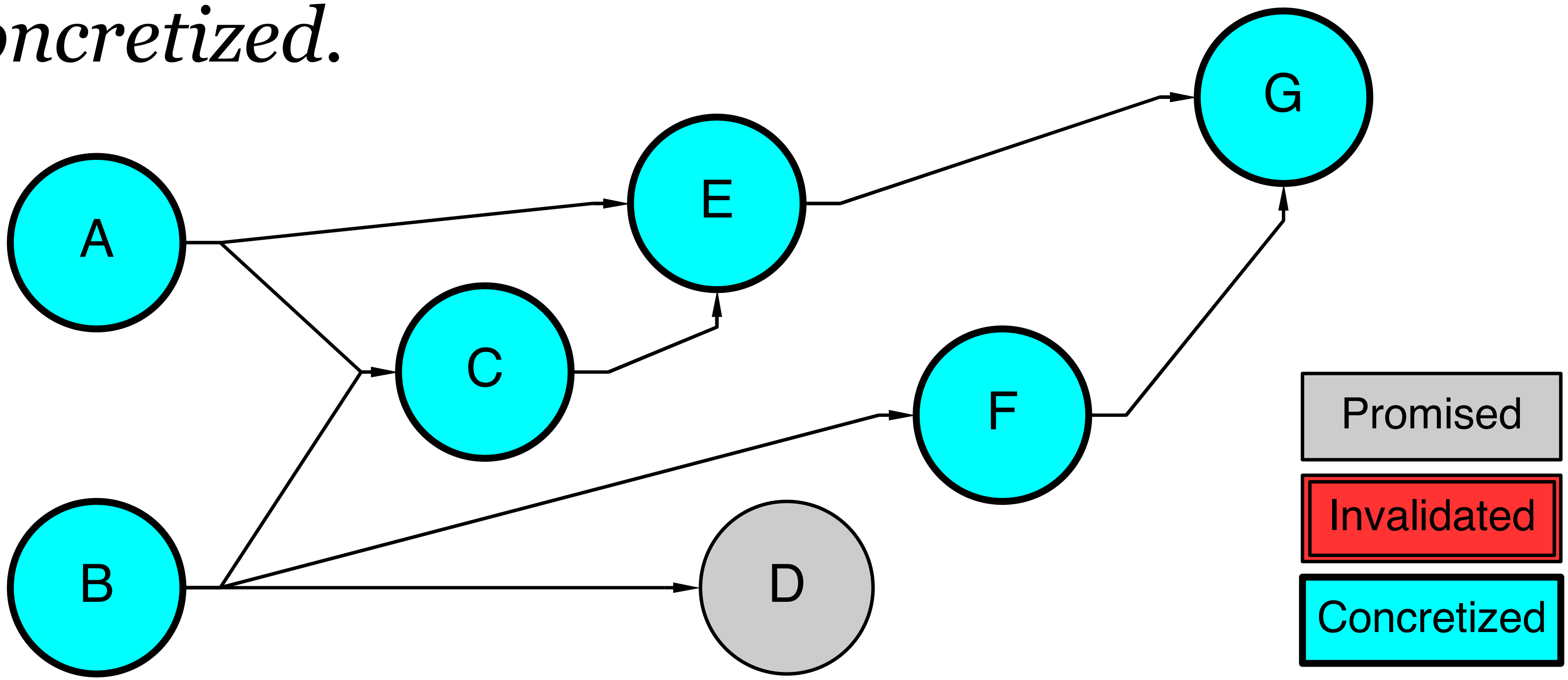
# Promises in a directed acyclic graph. *Each node has a value that is expensive to calculate and that depends on its parents.*




```
>>> create_graph('A->C; A->E; B->C; ...')
```

(A node holds a **`Promise`**, and pointers to parents and children)

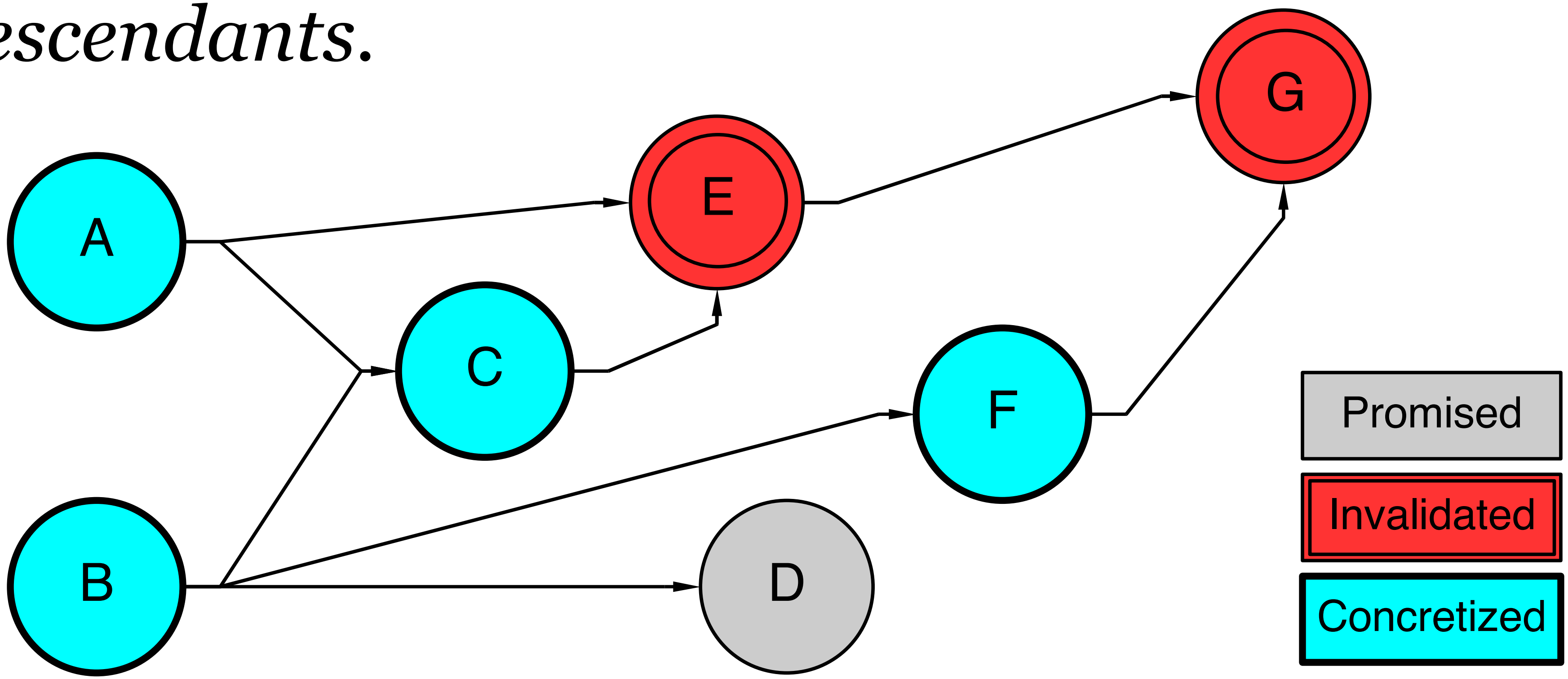
# Promises in a directed acyclic graph. *When a node is queried, its ancestors must be concretized.*




```
>>> query_value('G')
```

(A **`Promise`** is fulfilled by gaining a **`val`** attribute)

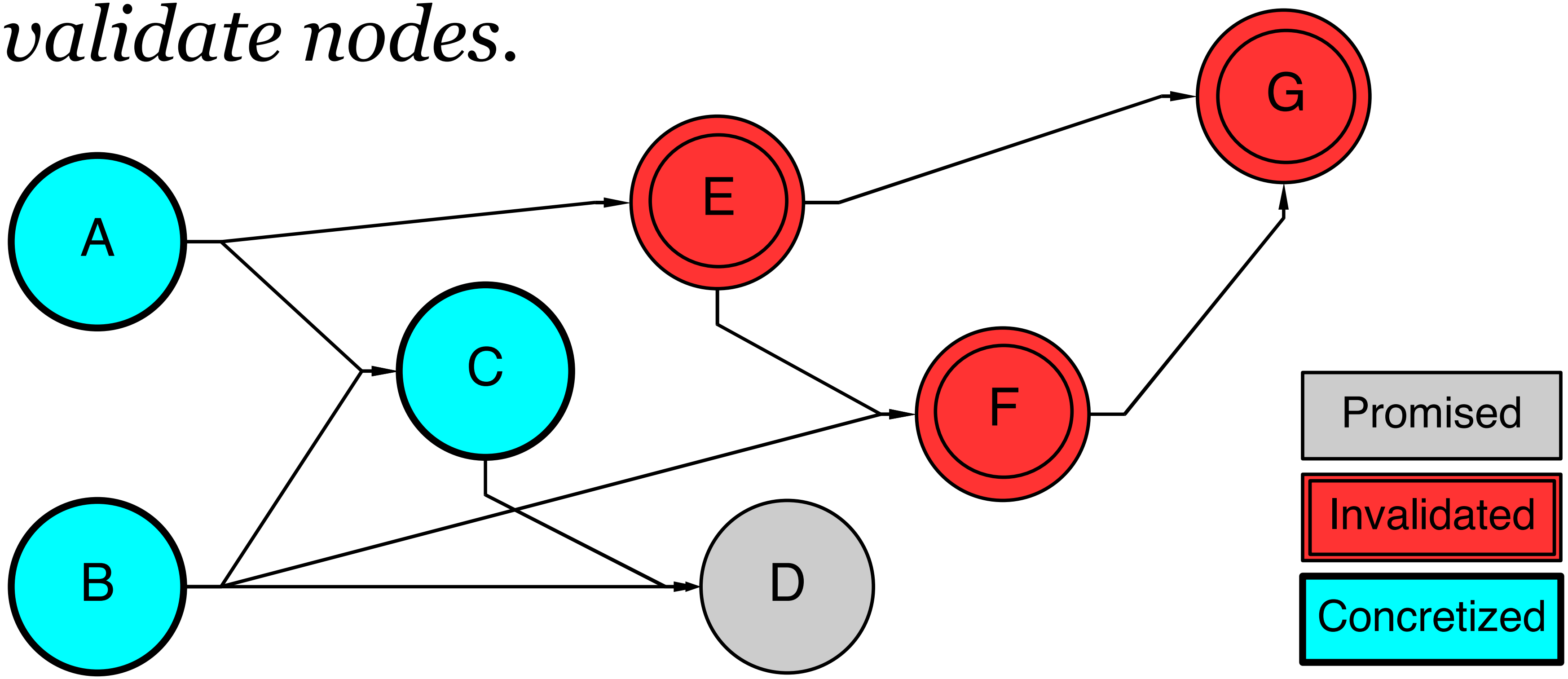
# Promises in a directed acyclic graph. *Changing the value of a node invalidates its descendants.*




```
>>> set_value('C')
```

(An invalid **Promise** might simply delete its **val** attribute)

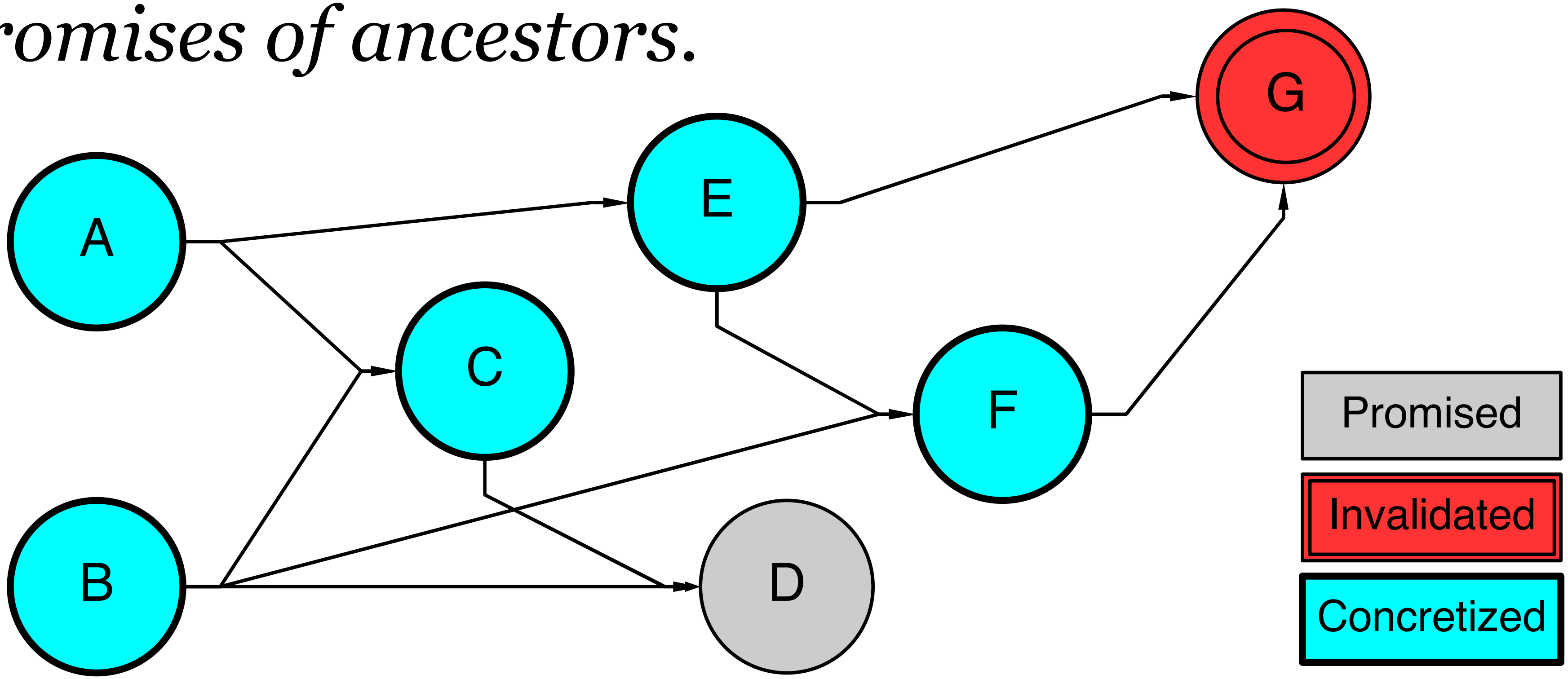
# Promises in a directed acyclic graph. *Changing the shape of a graph might invalidate nodes.*




```
>>> disconnect('C->E'); connect('E->F; C->D')
```

(Notice that `D` was unfulfilled, hence has no value to invalidate)

Promises in a directed acyclic graph. *Queries fulfill anew the previously invalidated promises of ancestors.*




```
>>> query_value('F')
```

# Wrap-up / Questions?

- *Review of laziness*
  - *Examples from Functional Programming Languages*
  - *Generic any language laziness*
- *Iterators and the `itertools` module*
- *Generators and generator expressions*
- *Memoization and the `weakref` module*
- *Laziness in a directed acyclic graph*