Better Code

Sean Parent | Principal Scientist

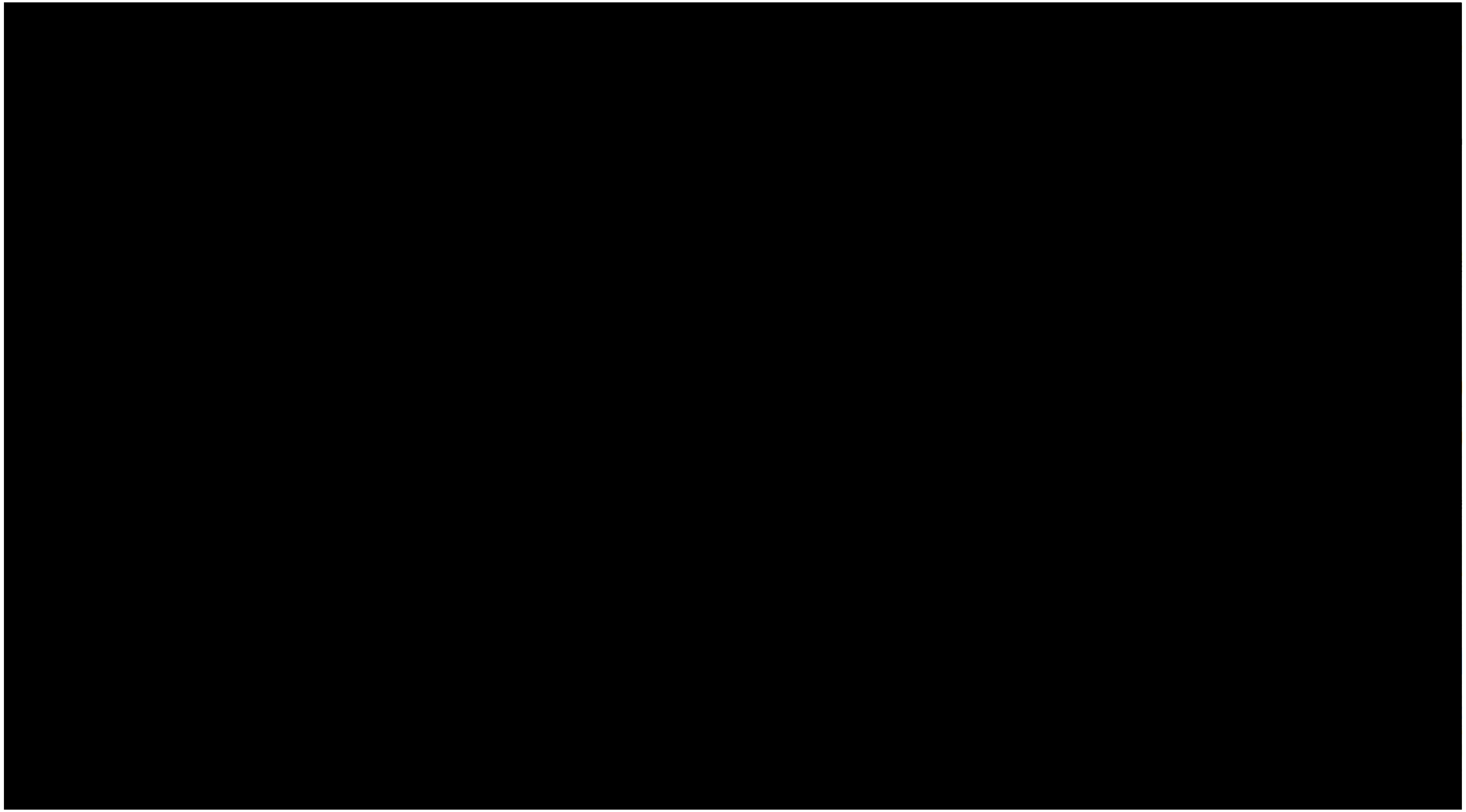
Bē
GMUNK

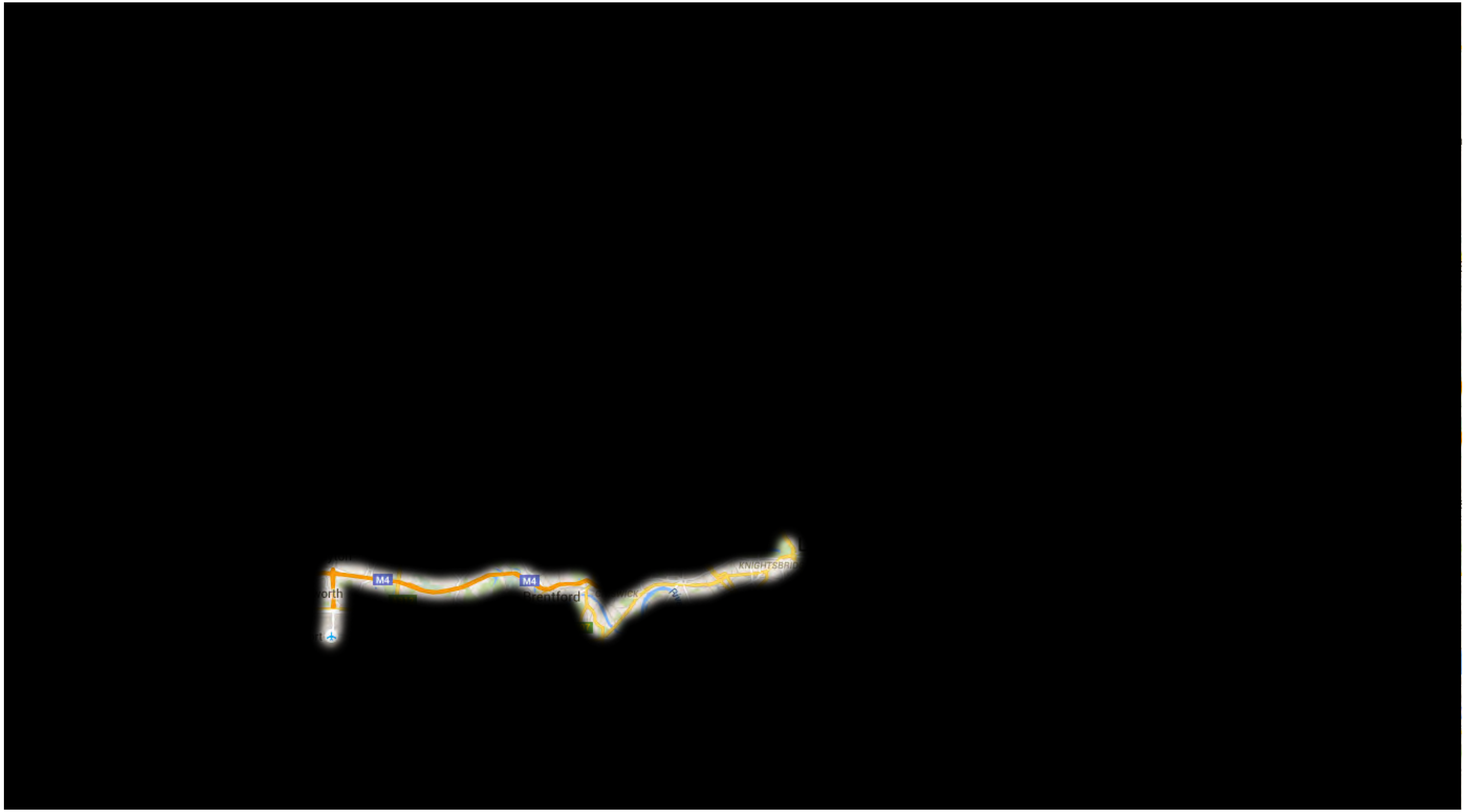
Better Code

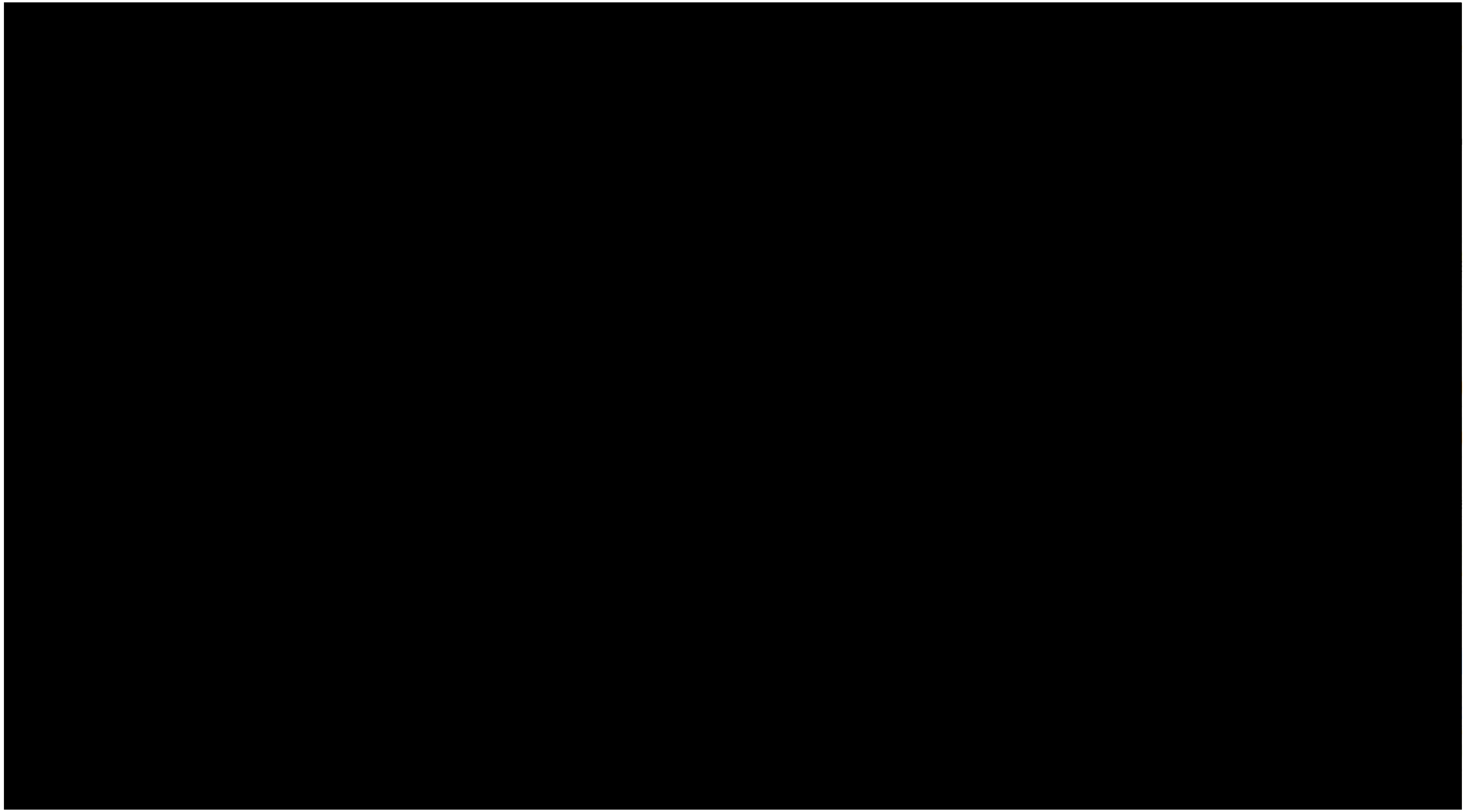
- Regular Types
  - Goal: No Incomplete Types
- Algorithms
  - Goal: No Raw Loops
- Data Structures
  - Goal: No Incidental Data Structures
- Runtime Polymorphism
  - Goal: No Inheritance
- Concurrency
  - Goal: No Raw Synchronization Primitives

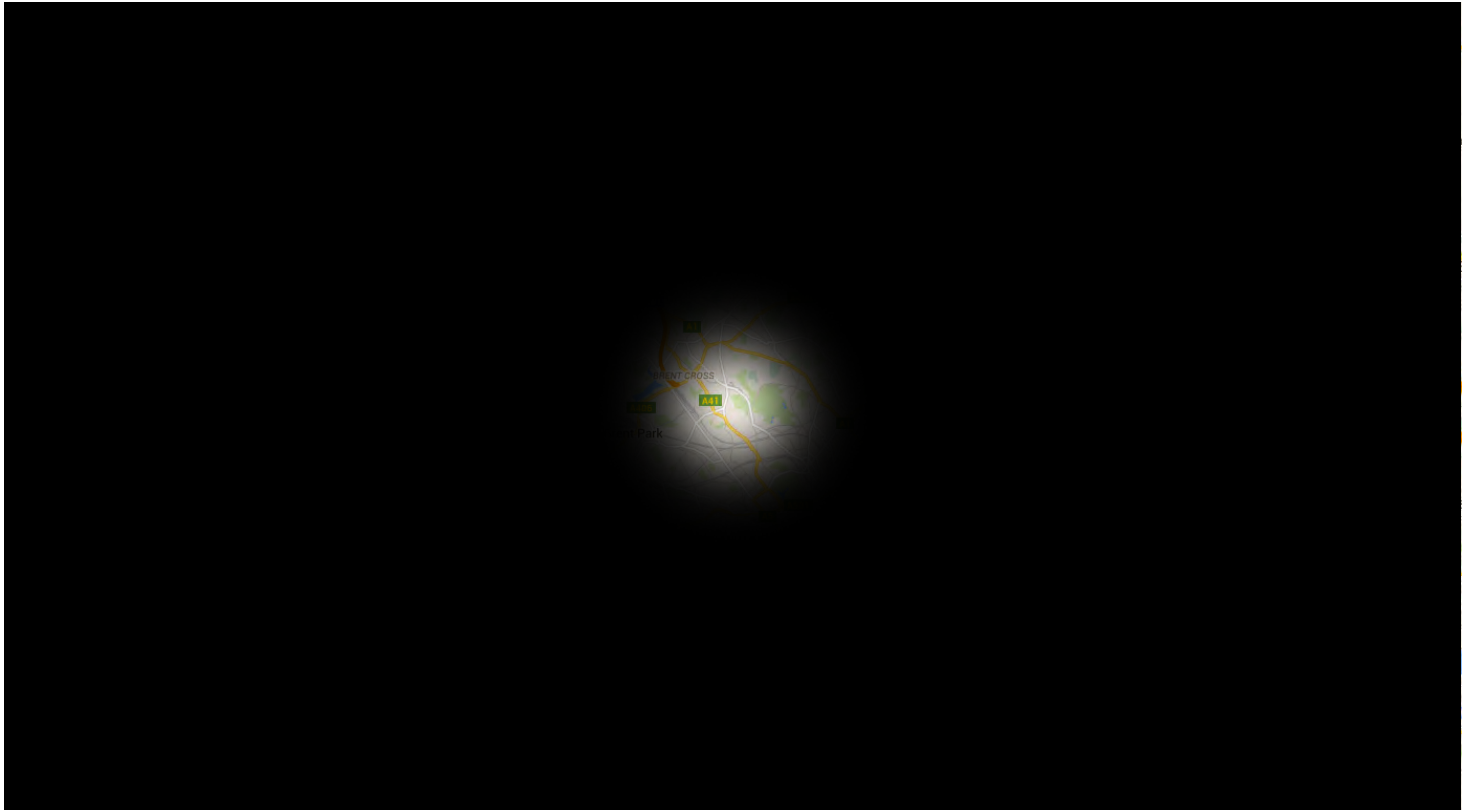
http://sean-parent.stlab.cc/papers-and-presentations

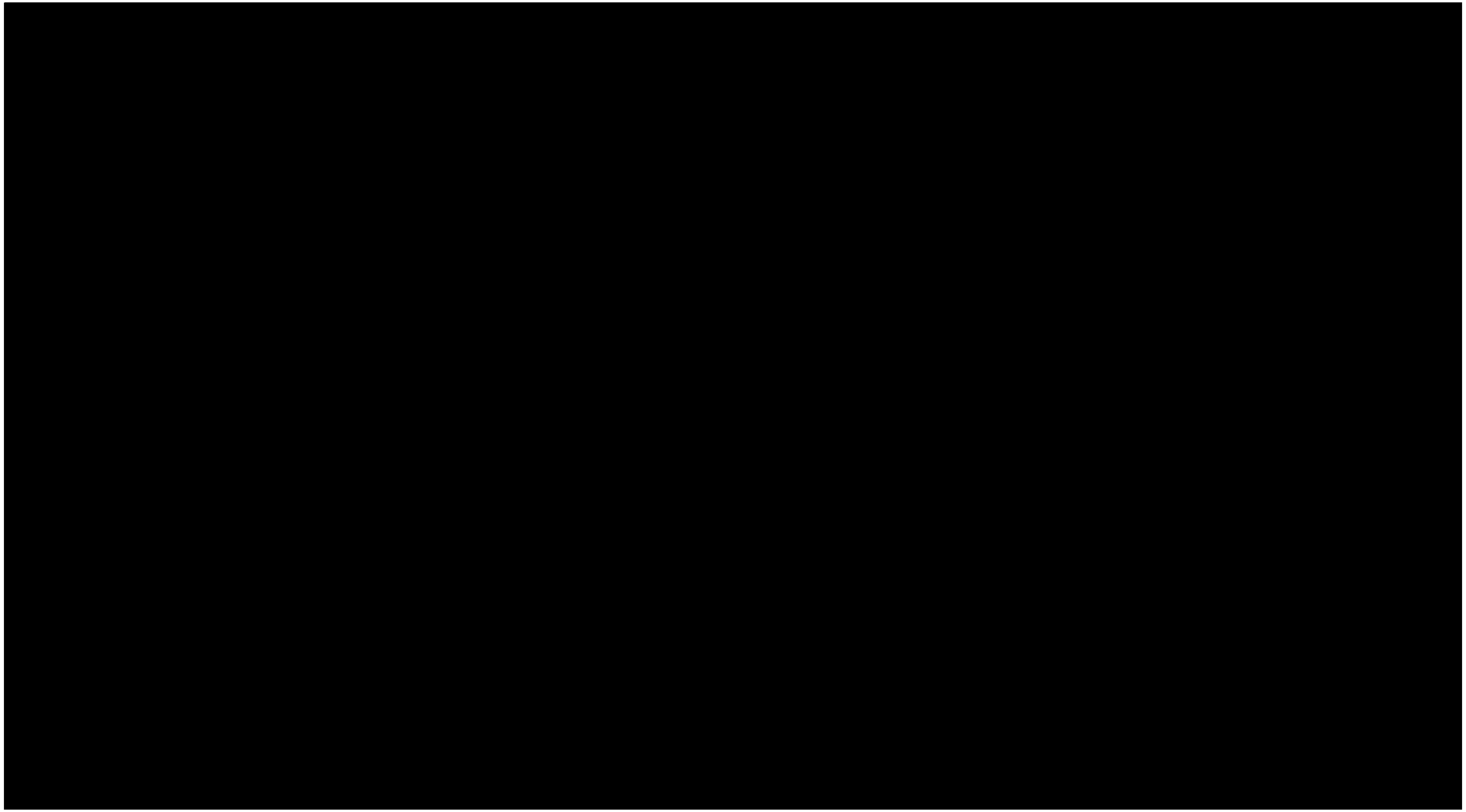
The Knowledge

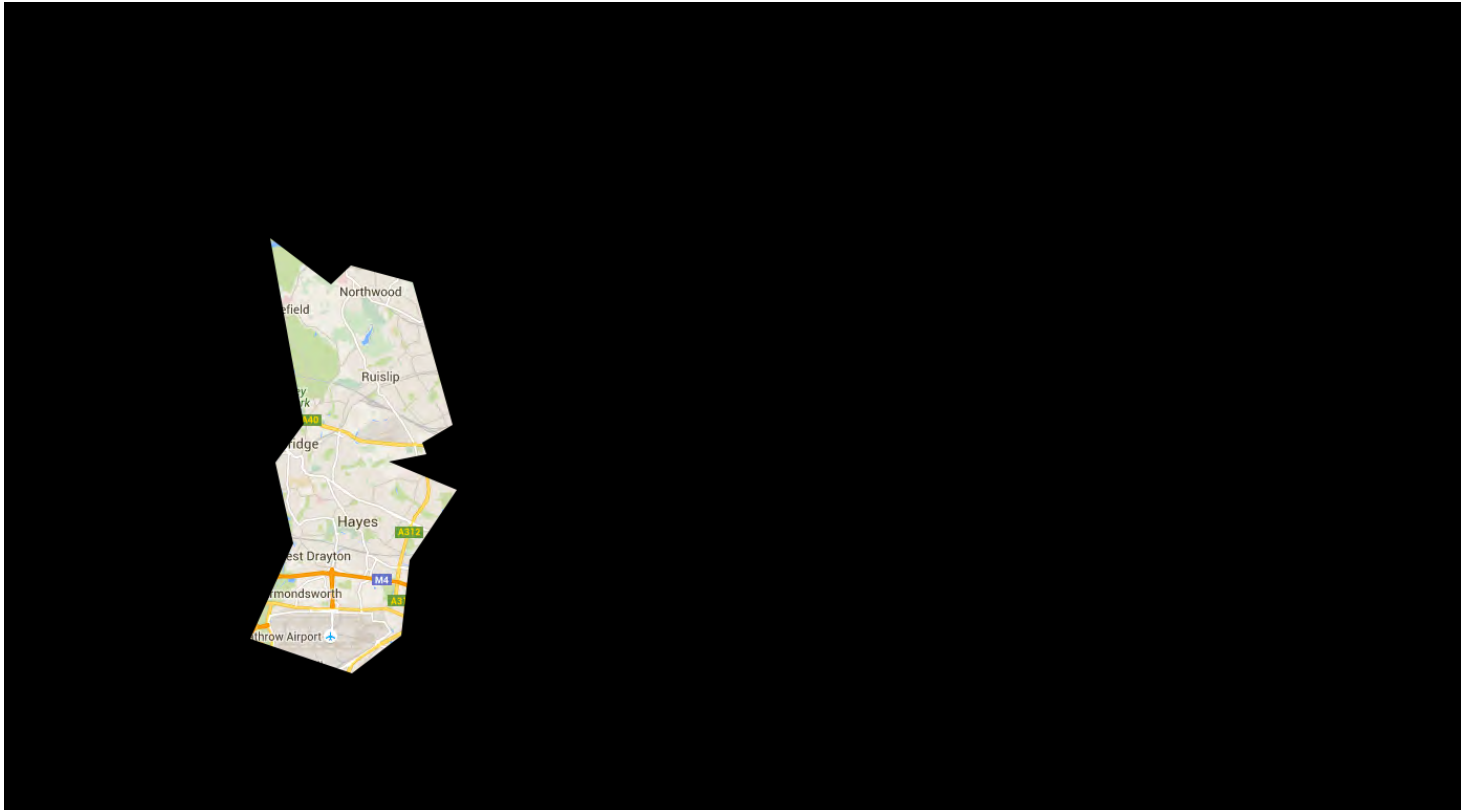


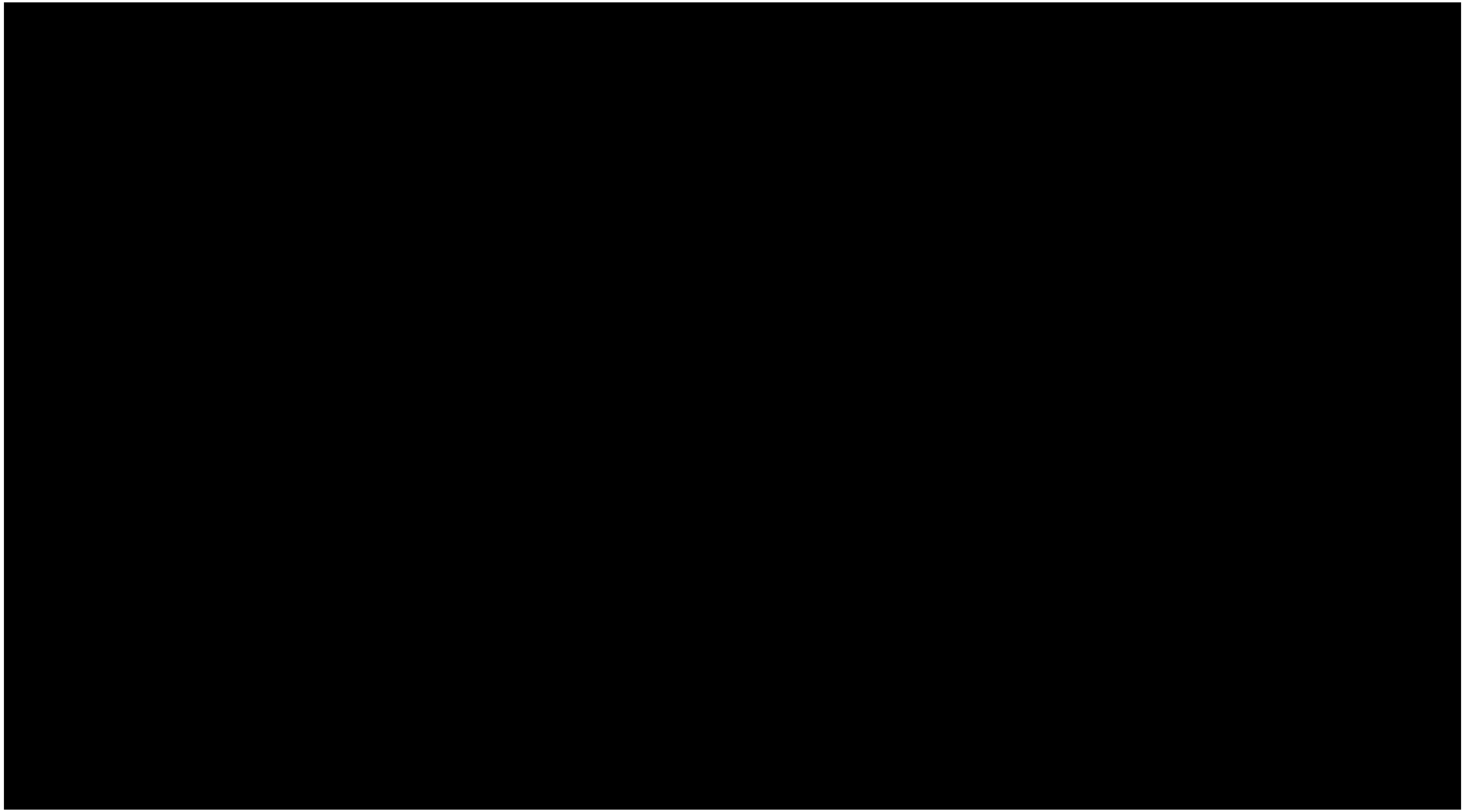

M4
M4
Brentford

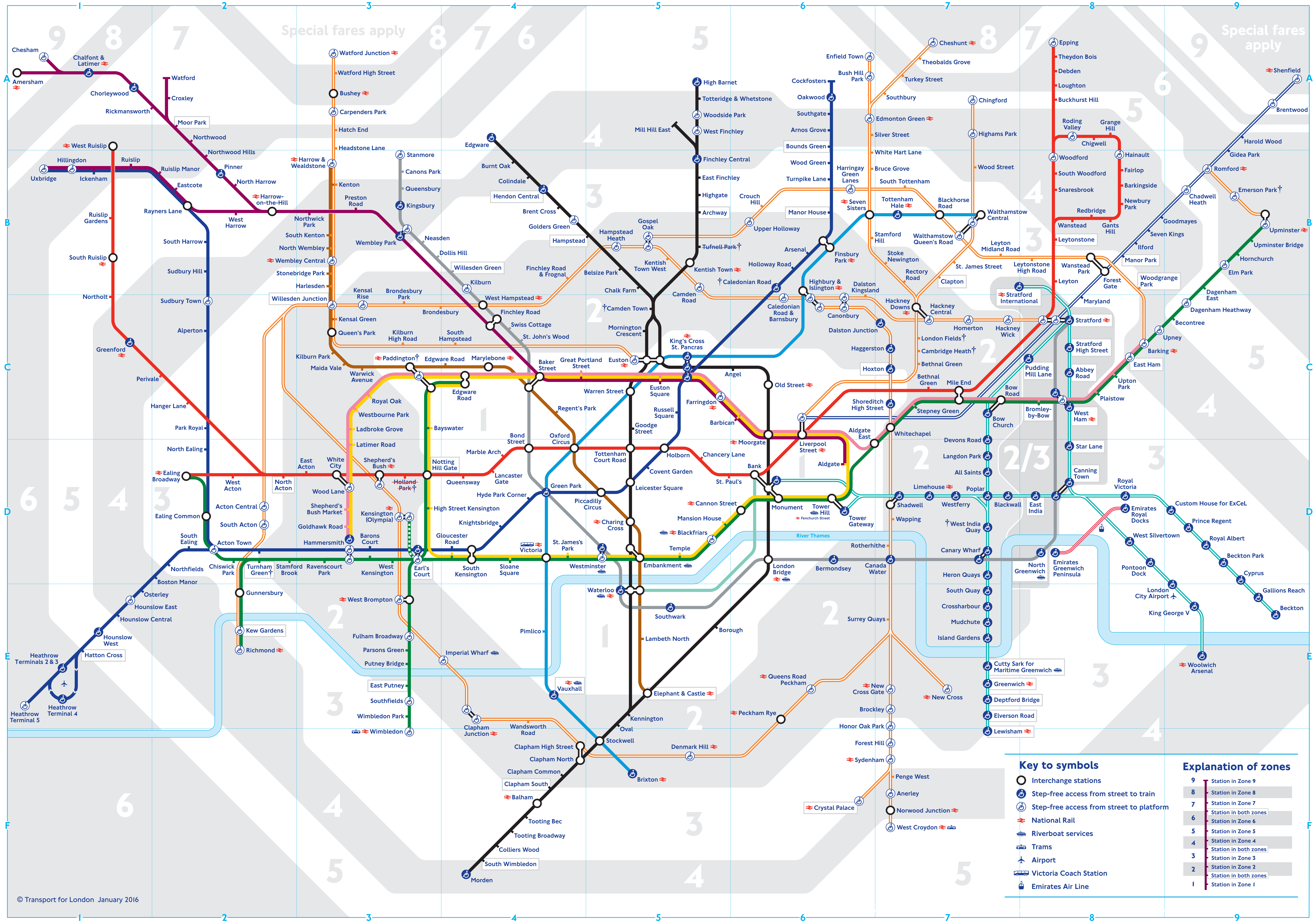



A41

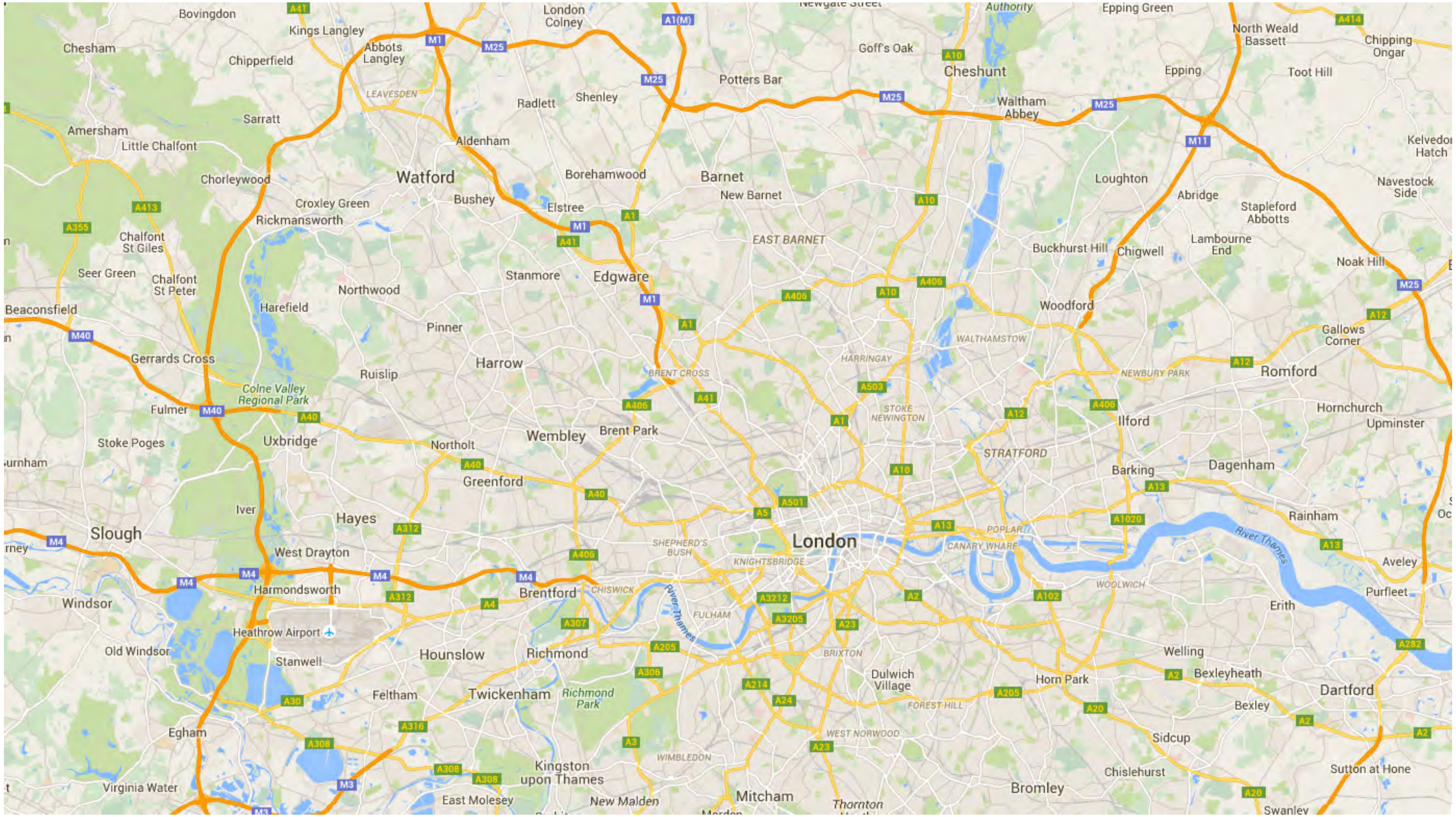



Northwood
Ruislip
A40
Hayes
A312
M4





Special fares apply
Special fares apply
Chesham
Amersham
Chalfont & Latimer
Chorleywood
Rickmansworth
Watford
Croxley
Moor Park
Northwood
Northwood Hills
Pinner
North Harrow
Harrow-on-the-Hill
West Harrow
West Ruislip
Ruislip
Hillingdon
Uxbridge
Ickenham
Ruislip Manor
Eastcote
Rayners Lane
Ruislip Gardens
South Harrow
South Ruislip
Sudbury Hill
Northolt
Sudbury Town
Alperton
Greenford
Perivale
Hanger Lane
Park Royal
North Ealing
Ealing Broadway
West Acton
North Acton
East Acton
Ealing Common
Acton Central
South Acton
Acton Town
South Ealing
Northfields
Boston Manor
Osterley
Hounslow East
Hounslow Central
Hounslow West
Hatton Cross
Heathrow Terminals 2 & 3
Heathrow Terminal 4
Heathrow Terminal 5
Chiswick Park
Turnham Green
Stamford Brook
Ravenscourt Park
Gunnersbury
Kew Gardens
Richmond
Watford Junction
Watford High Street
Bushey
Carpenders Park
Hatch End
Headstone Lane
Harrow & Wealdstone
Kenton
Preston Road
Northwick Park
South Kenton
North Wembley
Wembley Central
Stonebridge Park
Harlesden
Willesden Junction
Kensal Green
Queen's Park
Kilburn Park
Maida Vale
Warwick Avenue
Stanmore
Canons Park
Queensbury
Kingsbury
Wembley Park
Neasden
Dollis Hill
Willesden Green
Kilburn
West Hampstead
Kensal Rise
Brondesbury Park
Brondesbury
Kilburn High Road
South Hampstead
Finchley Road
Swiss Cottage
St. John's Wood
Edgware
Burnt Oak
Colindale
Hendon Central
Brent Cross
Golders Green
Hampstead
Finchley Road & Frognal
Hampstead Heath
Gospel Oak
Belsize Park
Chalk Farm
Kentish Town West
Camden Town
Camden Road
Mornington Crescent
Mill Hill East
High Barnet
Totteridge & Whetstone
Woodside Park
West Finchley
Finchley Central
East Finchley
Highgate
Archway
Crouch Hill
Tufnell Park
Kentish Town
Caledonian Road
Upper Holloway
Holloway Road
Cockfosters
Oakwood
Southgate
Arnos Grove
Bounds Green
Wood Green
Turnpike Lane
Harringay Green Lanes
Manor House
Seven Sisters
Arsenal
Finsbury Park
Highbury & Islington
Caledonian Road & Barnsbury
Canonbury
Dalston Kingsland
Dalston Junction
Haggerston
Hoxton
Enfield Town
Bush Hill Park
Edmonton Green
Silver Street
White Hart Lane
Bruce Grove
Tottenham Hale
South Tottenham
Cheshunt
Theobalds Grove
Turkey Street
Southbury
Stamford Hill
Stoke Newington
Rectory Road
Hackney Downs
Hackney Central
Clapton
Blackhorse Road
Walthamstow Central
Walthamstow Queen's Road
Leyton Midland Road
St. James Street
Chingford
Highams Park
Wood Street
Homerton
Hackney Wick
London Fields
Cambridge Heath
Bethnal Green
Epping
Theydon Bois
Debden
Loughton
Buckhurst Hill
Roding Valley
Grange Hill
Chigwell
Hainault
Fairlop
Barkingside
Newbury Park
Woodford
South Woodford
Snaresbrook
Redbridge
Wanstead
Gants Hill
Leytonstone
Leytonstone High Road
Wanstead Park
Manor Park
Forest Gate
Woodgrange Park
Leyton
Maryland
Stratford International
Stratford
Stratford High Street
Shenfield
Brentwood
Harold Wood
Gidea Park
Romford
Emerson Park
Chadwell Heath
Goodmayes
Seven Kings
Ilford
Upminster
Upminster Bridge
Hornchurch
Elm Park
Dagenham East
Dagenham Heathway
Becontree
Upney
Barking
East Ham
Upton Park
Plaistow
West Ham
Abbey Road
Pudding Mill Lane
Bow Road
Bow Church
Bromley-by-Bow
Mile End
Stepney Green
Star Lane
Canning Town
Royal Victoria
Custom House for ExCeL
Prince Regent
Royal Albert
Beckton Park
Cyprus
Gallions Reach
Beckton
Emirates Royal Docks
West Silvertown
Pontoon Dock
London City Airport
King George V
Woolwich Arsenal
Emirates Greenwich Peninsula
North Greenwich
Paddington
Edgware Road
Marylebone
Baker Street
Great Portland Street
Euston
King's Cross St. Pancras
Euston Square
Warren Street
Regent's Park
Russell Square
Goodge Street
Angel
Farringdon
Barbican
Old Street
Royal Oak
Westbourne Park
Ladbroke Grove
Latimer Road
Bayswater
White City
Shepherd's Bush
Wood Lane
Shepherd's Bush Market
Goldhawk Road
Hammersmith
Holland Park
Notting Hill Gate
Queensway
Lancaster Gate
Marble Arch
Bond Street
Oxford Circus
Tottenham Court Road
Holborn
Chancery Lane
Moorgate
Liverpool Street
Aldgate
Aldgate East
Shoreditch High Street
Whitechapel
Kensington (Olympia)
Barons Court
West Kensington
Earl's Court
High Street Kensington
Knightsbridge
Hyde Park Corner
Green Park
Piccadilly Circus
Covent Garden
Leicester Square
Charing Cross
Gloucester Road
South Kensington
Sloane Square
Victoria
St. James's Park
Westminster
Embankment
Temple
Blackfriars
Mansion House
Cannon Street
St. Paul's
Bank
Monument
Tower Hill
Fenchurch Street
Tower Gateway
Shadwell
Limehouse
Westferry
Poplar
Blackwall
East India
Devons Road
Langdon Park
All Saints
West India Quay
Canary Wharf
Heron Quays
South Quay
Crossharbour
Mudchute
Island Gardens
Cutty Sark for Maritime Greenwich
Greenwich
Deptford Bridge
Elverson Road
Lewisham
Wapping
Rotherhithe
Canada Water
Surrey Quays
Bermondsey
London Bridge
Southwark
Waterloo
Lambeth North
Borough
Elephant & Castle
Kennington
Oval
Stockwell
Vauxhall
Pimlico
Brixton
Clapham High Street
Clapham North
Clapham Common
Clapham South
Balham
Tooting Bec
Tooting Broadway
Colliers Wood
South Wimbledon
Morden
Wandsworth Road
Clapham Junction
Imperial Wharf
West Brompton
Fulham Broadway
Parsons Green
Putney Bridge
East Putney
Southfields
Wimbledon Park
Wimbledon
Denmark Hill
Peckham Rye
Queens Road Peckham
New Cross Gate
New Cross
Brockley
Honor Oak Park
Forest Hill
Sydenham
Penge West
Anerley
Crystal Palace
Norwood Junction
West Croydon
River Thames
© Transport for London January 2016
Key to symbols
Interchange stations
Step-free access from street to train
Step-free access from street to platform
National Rail
Riverboat services
Trams
Airport
Victoria Coach Station
Emirates Air Line
Explanation of zones
Station in Zone 9
Station in Zone 8
Station in Zone 7
Station in both zones
Station in Zone 6
Station in Zone 5
Station in Zone 4
Station in both zones
Station in Zone 3
Station in Zone 2
Station in both zones
Station in Zone 1


Bovingdon
Kings Langley
London Colney
Chesham
Chipperfield
Abbots Langley
Goff's Oak
Cheshunt
Epping Green
North Weald Bassett
Chipping Ongar
Potters Bar
Shenley
Radlett
Epping
Toot Hill
Waltham Abbey
Sarratt
Aldenham
Amersham
Little Chalfont
Watford
Borehamwood
Barnet
New Barnet
Loughton
Abridge
Stapleford Abbotts
Kelvedon Hatch
Navestock Side
Chorleywood
Croxley Green
Bushey
Elstree
Rickmansworth
Chalfont St Giles
EAST BARNET
Buckhurst Hill
Chigwell
Lambourne End
Noak Hill
Seer Green
Chalfont St Peter
Stanmore
Edgware
Northwood
Harefield
Woodford
Beaconsfield
Pinner
WALTHAMSTOW
Gallows Corner
Gerrards Cross
Harrow
HARRINGAY
Romford
Ruislip
BRENT CROSS
NEWBURY PARK
Colne Valley Regional Park
Fulmer
STOKE NEWINGTON
Hornchurch
Upminster
Ilford
Uxbridge
Wembley
Brent Park
Stoke Poges
Northolt
STRATFORD
Barking
Dagenham
Greenford
Iver
Hayes
Rainham
Slough
London
West Drayton
SHEPHERD'S BUSH
KNIGHTSBRIDGE
POPLAR
CANARY WHARF
Aveley
Harmondsworth
Windsor
Brentford
CHISWICK
WOOLWICH
Purfleet
Erith
Heathrow Airport
FULHAM
Old Windsor
Stanwell
Hounslow
Richmond
BRIXTON
Welling
Dulwich Village
Horn Park
Bexleyheath
Dartford
Feltham
Twickenham
Richmond Park
FOREST HILL
Bexley
Egham
WEST NORWOOD
Sidcup
WIMBLEDON
Kingston upon Thames
Chislehurst
Sutton at Hone
Virginia Water
East Molesey
New Malden
Mitcham
Thornton
Bromley
Swanley
River Thames
M1
M25
M4
M40
M11
M3
A1(M)
A41
A1
A10
A12
A13
A406
A40
A4
A2
A20
A23
A24
A3
A30
A312
A316
A308
A205
A306
A307
A214
A3205
A3212
A501
A5
A503
A102
A1020
A282
A355
A413
A414

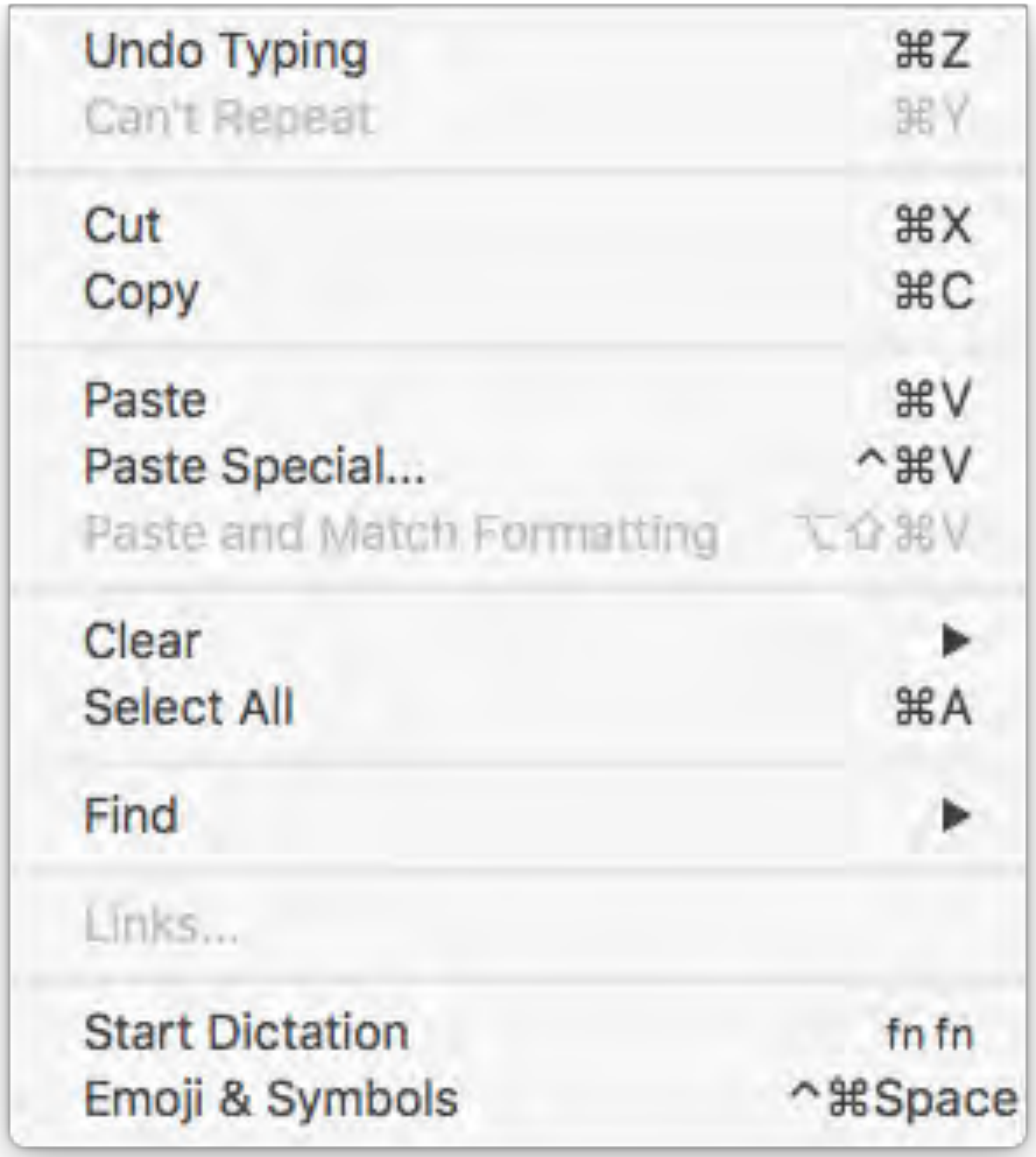

Bovingdon
Kings Langley
London Colney
Chesham
Chipperfield
Abbots Langley
LEAVESDEN
Goff's Oak
Cheshunt
Authority
Epping Green
North Weald Bassett
Chipping Ongar
Potters Bar
Shenley
Radlett
Sarratt
Amersham
Aldenham
Waltham Abbey
Epping
Toot Hill
Kelvedon Hatch
Watford
Borehamwood
Barnet
Loughton
Abridge
Stapleford Abbotts
Navestock Side
Bushey
Elstree
EAST BARNET
Chalfont St Giles
Seer Green
Chalfont St Peter
Stanmore
Edgware
Buckhurst Hill
Chigwell
Lambourne End
Noak Hill
Beaconsfield
Harefield
Gallows Corner
WALTHAMSTOW
Gerrards Cross
Colne Valley Regional Park
Ruislip
Romford
Fulmer
STOKE NEWINGTON
Hornchurch
Ilford
Upminster
Stoke Poges
Brent Park
STRATFORD
Dagenham
Greenford
Iver
Hayes
Rainham
Slough
West Drayton
Aveley
River Thames
Harmondsworth
Brentford
Purfleet
Windsor
WOOLWICH
Erith
Heathrow Airport
Old Windsor
Stanwell
Hounslow
Richmond
BRIXTON
Welling
Dulwich Village
Horn Park
Bexleyheath
Dartford
Feltham
Twickenham
Richmond Park
FOREST HILL
Bexley
Egham
WEST NORWOOD
Sidcup
Chislehurst
Sutton at Hone
Kingston upon Thames
WIMBLEDON
Mitcham
Bromley
Virginia Water
East Molesey
New Malden
Thornton
Swanley

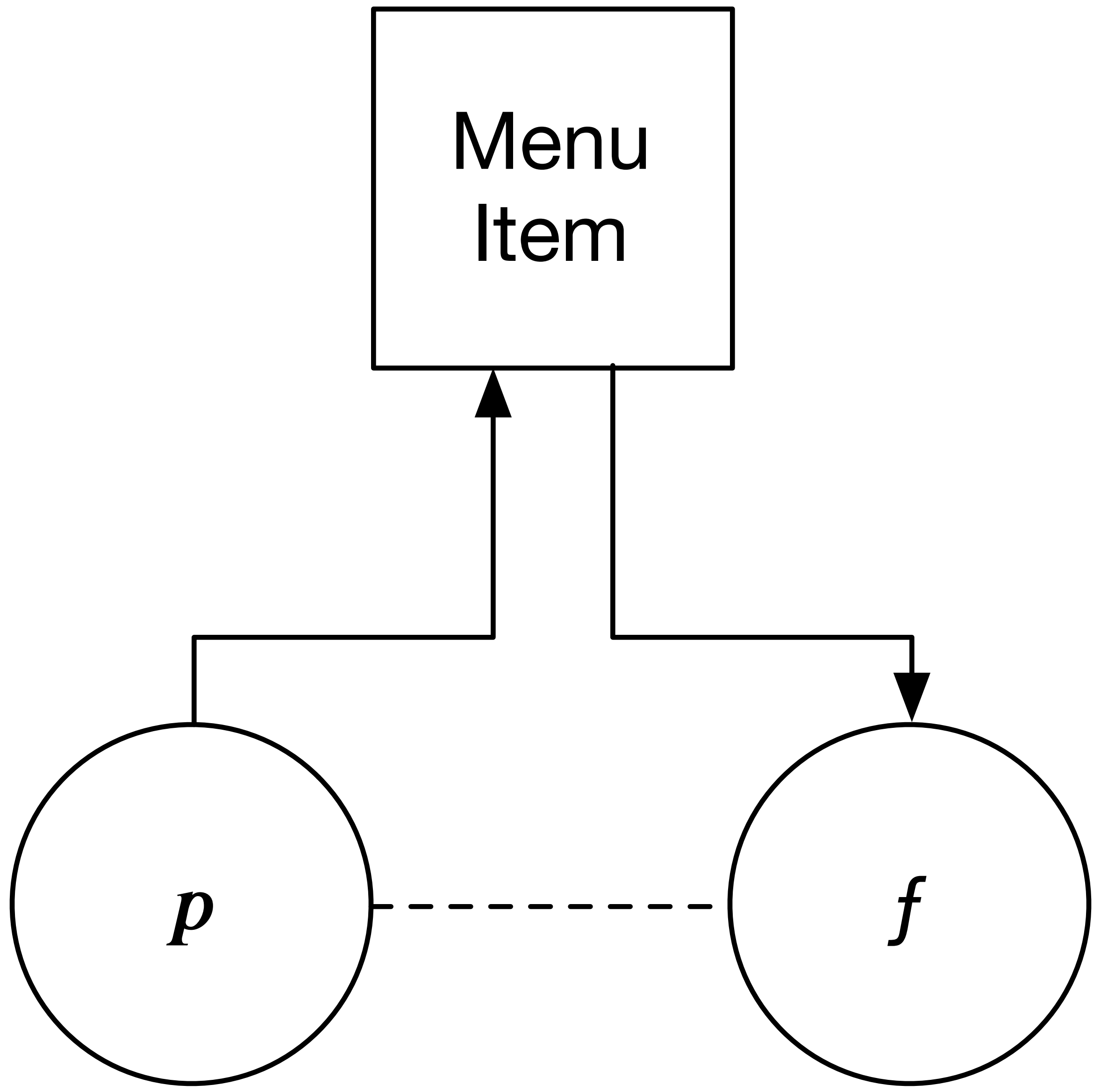
"There are rules!"

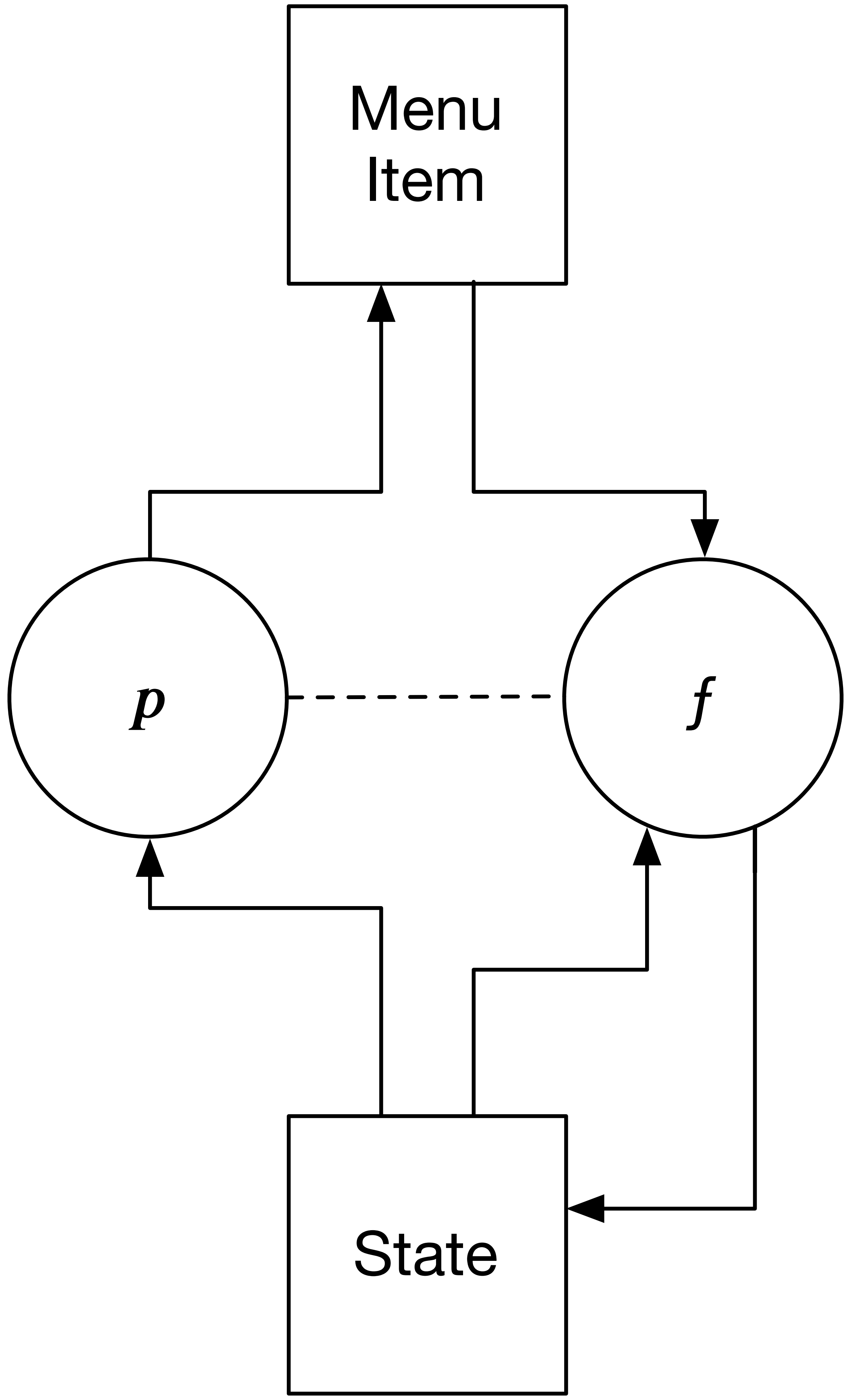
– The Big Lebowski

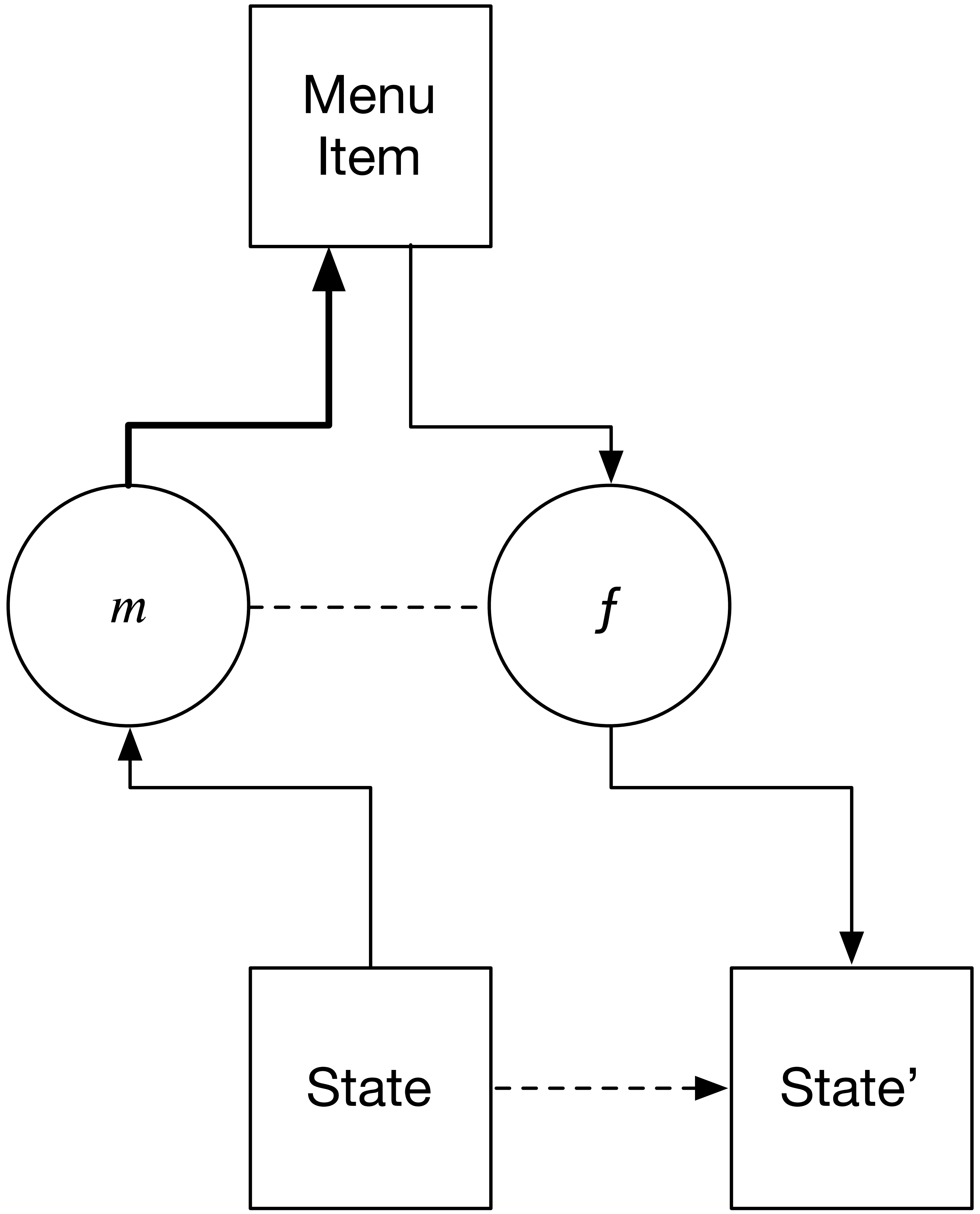
Lower Bound

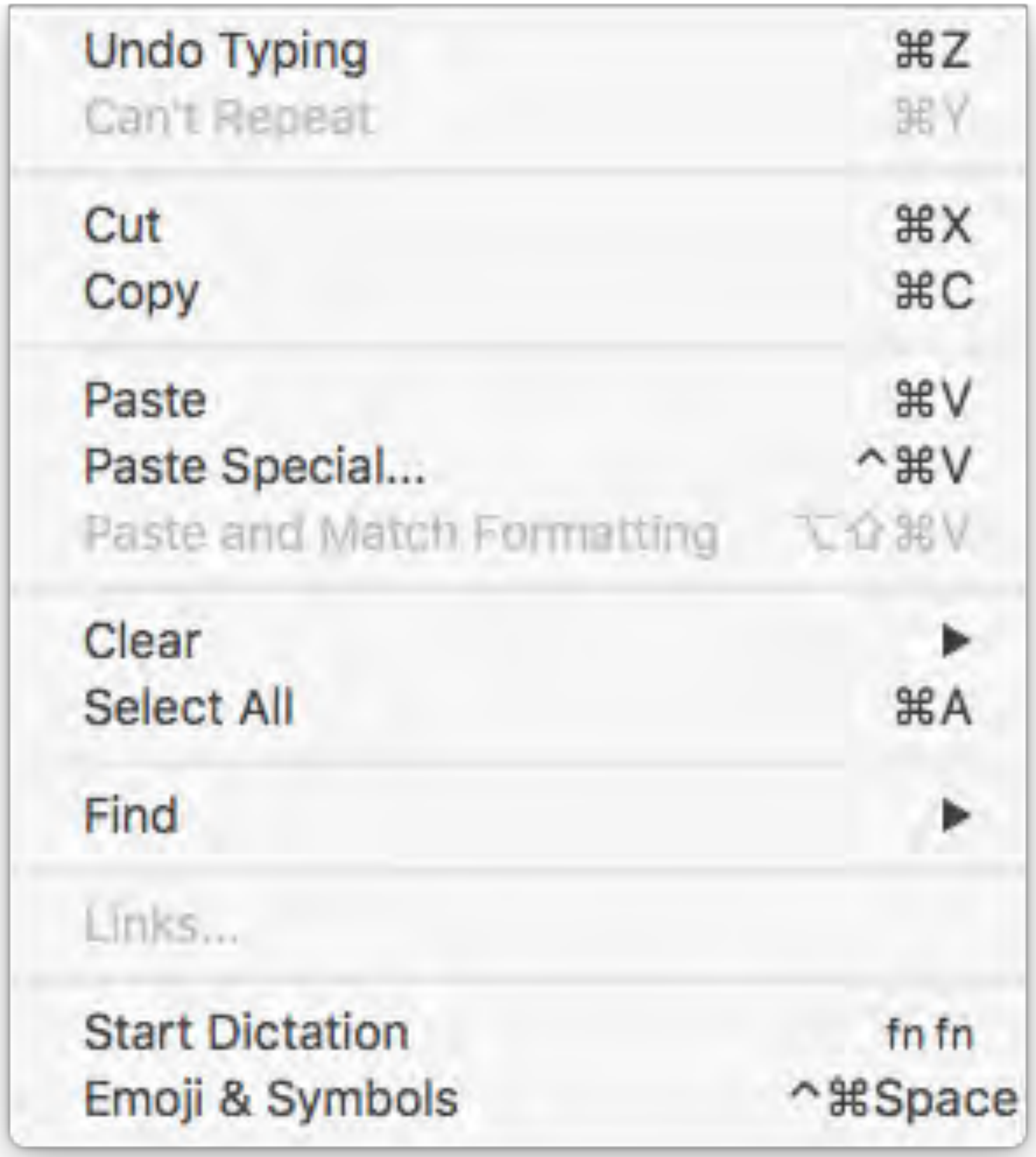
```
template <class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
        const T& value, Compare comp)
{
    auto n = distance(first, last);

    while (n != 0)  {
        auto h = n / 2;
        auto m = next(first, h);

        if (comp(*m, value)) {
            first = next(m);
            n -= h + 1;
        } else { n = h; }
    }

    return first;
}
```


Undo Typing ⌘Z
Can't Repeat ⌘Y
Cut ⌘X
Copy ⌘C
Paste ⌘V
Paste Special... ^⌘V
Paste and Match Formatting ⌥⇧⌘V
Clear ▶
Select All ⌘A
Find ▶
Links...
Start Dictation fn fn
Emoji & Symbols ^⌘Space


Menu
Item
p
f


Menu Item
p
f
State


Menu
Item
m
f
State
State’


Undo Typing ⌘Z
Can't Repeat ⌘Y
Cut ⌘X
Copy ⌘C
Paste ⌘V
Paste Special... ^⌘V
Paste and Match Formatting ⌥⇧⌘V
Clear ▶
Select All ⌘A
Find ▶
Links...
Start Dictation fn fn
Emoji & Symbols ^⌘Space

Good Code

Good code is *correct*

Good Code

Good code is *correct*

- Consistent; without contradiction

Simple Bug

```
void print_string(const char* s) {
    while (*s != '\0') {
        cout << *s++;
    }
}

int main() {
    print_string(nullptr);
}
```

Simple Bug

```
void print_string(const char* s) {
    while (*s != '\0') {                  Thread 1: EXC_BAD_ACCESS (code=1, address=0x0)
        cout << *s++;
    }
}

int main() {
    print_string(nullptr);
}
```

Simple Bug

```
void print_string(const char* s) {
    while (*s != '\0') {
        cout << *s++;
    }
}

int main() {
    print_string(nullptr);
}
```

Simple Bug

```
void print_string(const char* s) {
    while (*s != '\0') {
        cout << *s++;
    }
}

int main() {
    print_string(nullptr); // FORCE CRASH!
}
```

Subtle defects

Subtle defects

Consistency requires context

Subtle defects

Consistency requires context

```
template<class T> const T& min(const T& a, const T& b);
```

Returns: The smaller value.

Remarks: Returns the first argument when the arguments are equivalent.

Subtle defects

Consistency requires context

```
template<class T> const T& min(const T& a, const T& b);
```
Returns: The smaller value.

Remarks: Returns the first argument when the arguments are equivalent.

```
template<class T> const T& max(const T& a, const T& b);
```
Returns: The larger value.

Remarks: Returns the first argument when the arguments are equivalent.

Subtle defects

Subtle defects

```
template<typename T>
const T& clamp(const T& a, const T& lo, const T& hi)
{
    return min(max(lo, a), hi);
}
```

Subtle defects

```
template<typename T>
const T& clamp(const T& a, const T& lo, const T& hi)
{
    return min(max(lo, a), hi);
}

template<typename T, typename Compare>
const T& clamp(const T& a, const T& lo, const T& hi, Compare comp)
{
    return min(max(lo, a, comp), hi, comp);
}
```

Subtle defects

Subtle defects

```
int main() {
    using pair = pair<int, string>;

    pair a = { 1, "OK" };

    pair lo = { 1, "FAIL: LO" };
    pair hi = { 2, "FAIL: HI" };

    a = clamp(a, lo, hi, [](const auto& a, const auto& b) {
        return a.first < b.first;
    });

    cout << a.second << endl;
};
```

Subtle defects

```
int main() {
    using pair = pair<int, string>;

    pair a = { 1, "OK" };

    pair lo = { 1, "FAIL: LO" };
    pair hi = { 2, "FAIL: HI" };

    a = clamp(a, lo, hi, [](const auto& a, const auto& b) {
        return a.first < b.first;
    });

    cout << a.second << endl;
};
```

FAIL: LO

Subtle defects

Subtle defects

```
template<typename T>
const T& clamp(const T& a, const T& lo, const T& hi)
{
    return min(max(a, lo), hi);
}
```

Subtle defects

```
template<typename T>
const T& clamp(const T& a, const T& lo, const T& hi)
{
    return min(max(a, lo), hi);
}

template<typename T, typename Compare>
const T& clamp(const T& a, const T& lo, const T& hi, Compare comp)
{
    return min(max(a, lo, comp), hi, comp);
}
```

Subtle defects

Subtle defects

```
template<class T> const T& min(const T& a, const T& b);
```
Returns: The smaller value.

Remarks: Returns the first argument when the arguments are equivalent.

```
template<class T> const T& max(const T& a, const T& b);
```
Returns: The larger value.

Remarks: Returns the **second** argument when the arguments are equivalent.

Subtle defects

```
template<class T> const T& min(const T& a, const T& b);
```
Returns: The smaller value.

Remarks: Returns the first argument when the arguments are equivalent.

```
template<class T> const T& max(const T& a, const T& b);
```
Returns: The larger value.

Remarks: Returns the **second** argument when the arguments are equivalent.

```
template <class T> const T& max(const T& a, const T& b, const T& c);
```
Returns: The larger value.

Remarks: **???**

Rules are Contentious

Rules are Contentious

"Names should not be associated with semantics because everybody has their own hidden assumptions about what semantics are, and they clash, causing comprehension problems without knowing why. This is why it's valuable to write code to reflect what code is actually doing, rather than what code 'means': it's hard to have conceptual clashes about what code actually does."

– Craig Silverstein, Google


Adobe
“There is no spoon.”
– The Matrix
Bē
GMUNK

How can nothing be something?

How can nothing be something?

```
int x;
```

How can nothing be something?

```
int x;
 // indeterminate value
```

How can nothing be something?

```
int x;
 // indeterminate value

int x = 1 / 0;
```

How can nothing be something?

```
int x;
 // indeterminate value

int x = 1 / 0;
 // undefined behavior
```

How can nothing be something?

```
int x;
 // indeterminate value

int x = 1 / 0;
 // undefined behavior

double x = 1.0 / 0.0;
```

How can nothing be something?

```
int x;
 // indeterminate value

int x = 1 / 0;
 // undefined behavior

double x = 1.0 / 0.0;
 // inf
```

How can nothing be something?

```
int x;
 // indeterminate value

int x = 1 / 0;
 // undefined behavior

double x = 1.0 / 0.0;
 // inf

double x = 0.0 / 0.0;
```

How can nothing be something?

```
int x;
 // indeterminate value

int x = 1 / 0;
 // undefined behavior

double x = 1.0 / 0.0;
 // inf

double x = 0.0 / 0.0;
 // NaN
```

How can nothing be something?

```
int x;
 // indeterminate value

int x = 1 / 0;
 // undefined behavior

double x = 1.0 / 0.0;
 // inf

double x = 0.0 / 0.0;
 // NaN

struct empty { };
```

How can nothing be something?

```
int x;
 // indeterminate value

int x = 1 / 0;
 // undefined behavior

double x = 1.0 / 0.0;
 // inf

double x = 0.0 / 0.0;
 // NaN

struct empty { };
 // sizeof(empty) == 1
```

How can nothing be something?

How can nothing be something?

```
int a[0];
```

How can nothing be something?

```
int a[0];
 // Error
```

How can nothing be something?

```
int a[0];
 // Error
 // but common extension
```

How can nothing be something?

```
int a[0];
 // Error
 // but common extension
 using empty = int[0];
```

How can nothing be something?

```
int a[0];
 // Error
 // but common extension
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
```

How can nothing be something?

```
int a[0];
 // Error
 // but common extension
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]
```

How can nothing be something?

```
int a[0];
 // Error
 // but common extension
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]

void f() { return void(); }
```

How can nothing be something?

```
int a[0];
 // Error
 // but common extension
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]

void f() { return void(); }
 // OK
```

How can nothing be something?

```
int a[0];
 // Error
 // but common extension
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]

void f() { return void(); }
 // OK

void x = f();
```

How can nothing be something?

```
int a[0];
 // Error
 // but common extension
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]

void f() { return void(); }
 // OK

void x = f();
 // Error
```

How can nothing be something?

```
int a[0];
 // Error
 // but common extension
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]

void f() { return void(); }
 // OK

void x = f();
 // Error
 // but void* is a pointer to anything…
```

How can nothing be something?

How can nothing be something?

```
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
```

How can nothing be something?

```
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
 // Basic Exception Guarantee:
 // Valid but unspecified
```

How can nothing be something?

```
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
 // Basic Exception Guarantee:
 // Valid but unspecified

std::vector<int> y = std::move(x);
```

How can nothing be something?

```
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
 // Basic Exception Guarantee:
 // Valid but unspecified

std::vector<int> y = std::move(x);
 // Moved from object, x, is valid but unspecified
```

Good Code

Good Code

Good code is *correct*

- Consistent; without contradiction

Good Code

Good code is *correct*

- Consistent; without contradiction

Good code has *meaning*

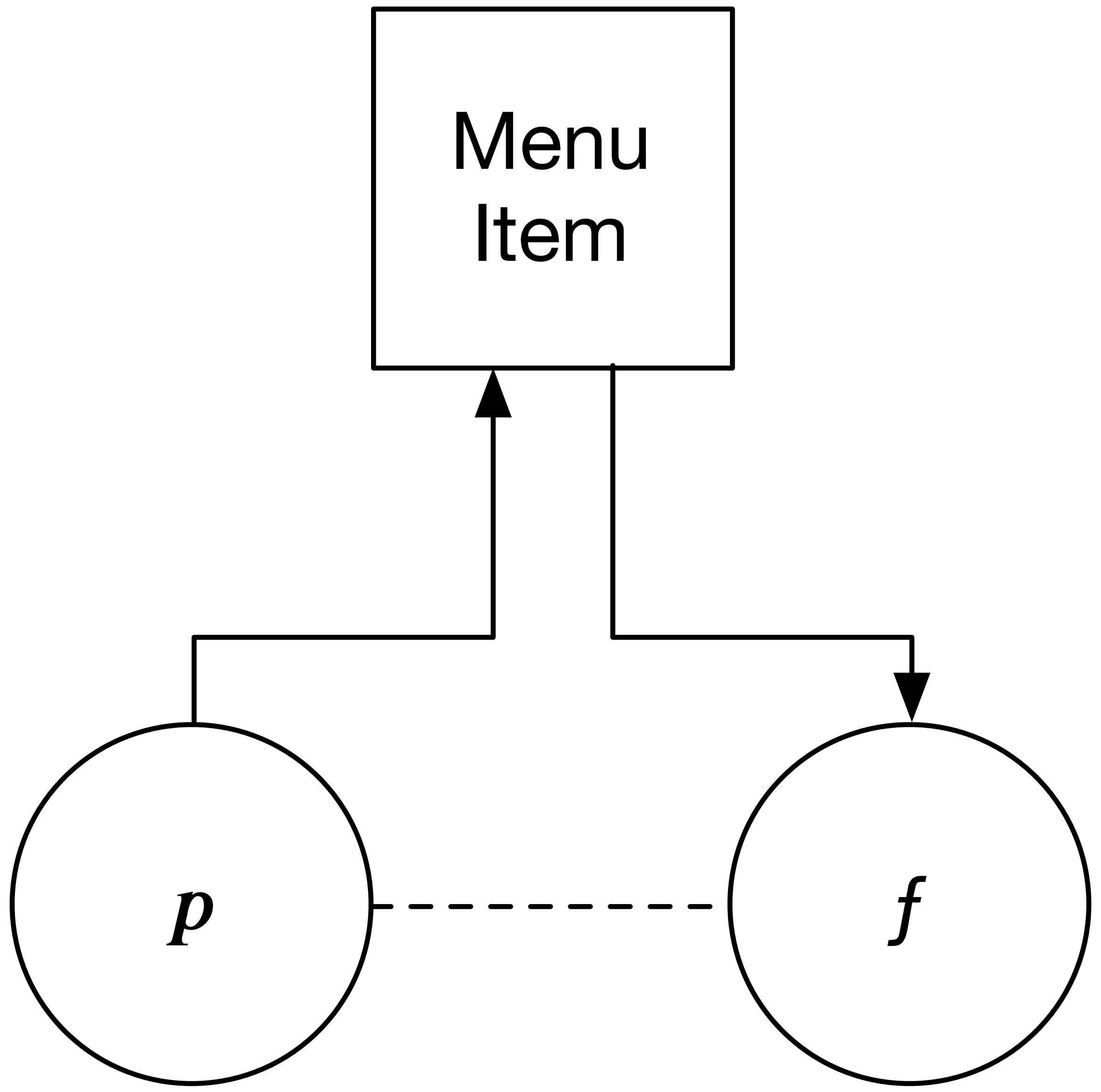
Good Code

Good code is *correct*

Consistent; without contradiction

Good code has *meaning*

Correspondence to an entity; specified, defined


Menu
Item
p
f

Categories of nothing

Categories of nothing

Absence of *something*

0, Ø, [p, p), void

Categories of nothing

Absence of *something*

0, $\emptyset$, $[p, p)$, void

Absence of *meaning*

NaN, undefined, indeterminate

How can nothing be something?

How can nothing be something?

```
int x;
```

How can nothing be something?

```
int x;
 // Partially formed; assign value or destruct
```

How can nothing be something?

```
int x;
 // Partially formed; assign value or destruct

int x = 1 / 0;
```

How can nothing be something?

```
int x;
 // Partially formed; assign value or destruct

int x = 1 / 0;
 // undefined behavior; reading from meaningless value
```

How can nothing be something?

```
int x;
 // Partially formed; assign value or destruct

int x = 1 / 0;
 // undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
```

How can nothing be something?

```
int x;
 // Partially formed; assign value or destruct

int x = 1 / 0;
 // undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
 // inf; OK, approximation for underflow
```

How can nothing be something?

```
int x;
 // Partially formed; assign value or destruct

int x = 1 / 0;
 // undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
 // inf; OK, approximation for underflow

double x = 0.0 / 0.0;
```

How can nothing be something?

```
int x;
 // Partially formed; assign value or destruct

int x = 1 / 0;
 // undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
 // inf; OK, approximation for underflow

double x = 0.0 / 0.0;
 // NaN; OK, though undefined behavior would also be
```

How can nothing be something?

```
int x;
 // Partially formed; assign value or destruct

int x = 1 / 0;
 // undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
 // inf; OK, approximation for underflow

double x = 0.0 / 0.0;
 // NaN; OK, though undefined behavior would also be
```

How can nothing be something?

```
int x;
 // Partially formed; assign value or destruct

int x = 1 / 0;
 // undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
 // inf; OK, approximation for underflow

double x = 0.0 / 0.0;
 // NaN; OK, though undefined behavior would also be

struct empty : void { };
```

How can nothing be something?

```
int x;
 // Partially formed; assign value or destruct

int x = 1 / 0;
 // undefined behavior; reading from meaningless value

double x = 1.0 / 0.0;
 // inf; OK, approximation for underflow

double x = 0.0 / 0.0;
 // NaN; OK, though undefined behavior would also be

struct empty : void { };
 // sizeof(empty) == 0;
```

How can nothing be something?

How can nothing be something?

```
int a[0];
```

How can nothing be something?

```
int a[0];
 // OK
```

How can nothing be something?

```
int a[0];
 // OK
 using empty = int[0];
```

How can nothing be something?

```
int a[0];
 // OK
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
```

How can nothing be something?

```
int a[0];
 // OK
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]
```

How can nothing be something?

```
int a[0];
 // OK
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]

void f() { return void(); }
```

How can nothing be something?

```
int a[0];
 // OK
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]

void f() { return void(); }
 // OK
```

How can nothing be something?

```
int a[0];
 // OK
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]

void f() { return void(); }
 // OK

void x = f();
```

How can nothing be something?

```
int a[0];
 // OK
 using empty = int[0];
 // sizeof(empty) == 0
 empty a[2];
 // &a[0] == &a[1]

void f() { return void(); }
 // OK

void x = f();
 // OK
 // void* is OK
```

How can nothing be something?

How can nothing be something?

```
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
```

How can nothing be something?

```
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
 // Basic Exception Guarantee:
 // Partially formed object. Reading is undefined behavior
```

How can nothing be something?

```
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
 // Basic Exception Guarantee:
 // Partially formed object. Reading is undefined behavior

std::vector<int> y = std::move(x);
```

How can nothing be something?

```
std::vector<int> x = { 1, 2, 3 };
try {
    x.insert(x.begin(), 0);
} catch (...) {
    std::cout << x.size() << std::endl;
}
 // Basic Exception Guarantee:
 // Partially formed object. Reading is undefined behavior

std::vector<int> y = std::move(x);
 // Moved from object, x, is partially formed
```

“That makes you wonder. Take chicken, for example.”

– Matrix

Specification

Specification

- `clone_ptr<T>` is like `std::unique_ptr<T>` but with two additional operations, copy and assignment that copy the object pointed to.

Specification

- `clone_ptr<T>` is like `std::unique_ptr<T>` but with two additional operations, copy and assignment that copy the object pointed to.

- Example implementation of new operations:

```
clone_ptr(const clone_ptr& x) : _ptr(new T(*x)) { }
clone_ptr& operator=(const clone_ptr& x) { return *this = clone_ptr(x); }
```

- copy-assignment written in terms of copy and move-assignment

What is copy?

- *Copying* an object creates a new object which is equal-to and logically disjoint from the original.

```
T a = b; ⇒ a == b;
T a = b; modify(b); ⇒ a != b;
```

"copy" of clone_ptr

```
clone_ptr<T> a = b; ⇒ a != b;
```

- *"Copying"* a clone pointer creates an object that is not equal to the original
- Contradiction

- Defining a copy-constructor that doesn't copy is dangerous
  - The compiler may elide copies
  - Programmers will assume they are substitutable

Specification: Amendment 1

- Two `clone_ptrs` are considered equal if the value they point to is equal. Because we don't want to require that the pointed to types are equal `operator==()` and `operator!=()` are not implemented. i.e.:

```
clone_ptr<T> a = b; ⇒ a == b;
```

However, == is not implemented.

What is a pointer?

- A *pointer* is an object that refers to another object via a dereference operation. Two pointers are equal if they refer to the same instance of an object.

```
a == b; ⇒ &*a == &*b;
```

“equality” of clone_ptr

```
clone_ptr<T> a = b; ⇒ a == b;
```

- Because `clone_ptr` is a pointer this would imply:

```
assert(&*a == &*b);
```

- But that is false - contradiction.

Specification: Amendment 2

- Because `clone_ptr<>` is not a pointer it is to be renamed `indirect<>`.

What is a const?

- *const* is a reference modify. An object accessed through a *const* reference may not be modified.

```
const T a = b; read(a); ⇒ a == b;
modify(a); is not allowed
```

“const” of indirect

```
const indirect<T> a = b; read(a); ⇏ a == b;
```

- Because const does not propagate (from unique_ptr):

```
void read(const indirect<T>& x) {
    modify(*x);
}
```

- Contradiction!

Specification: Amendment 3

- Because copy of remote part implies const propagation, get(), operator*() and operator->() must be overloaded:

```
T* get();
const T* get() const;

T& operator*();
const T& operator*() const;

T* operator->();
const T* operator->() const;
```

Alternative Specification:

- `clone_ptr<T>` is like `std::unique_ptr<T>` but with one additional operation, `clone()` that works by copying the object pointed to.

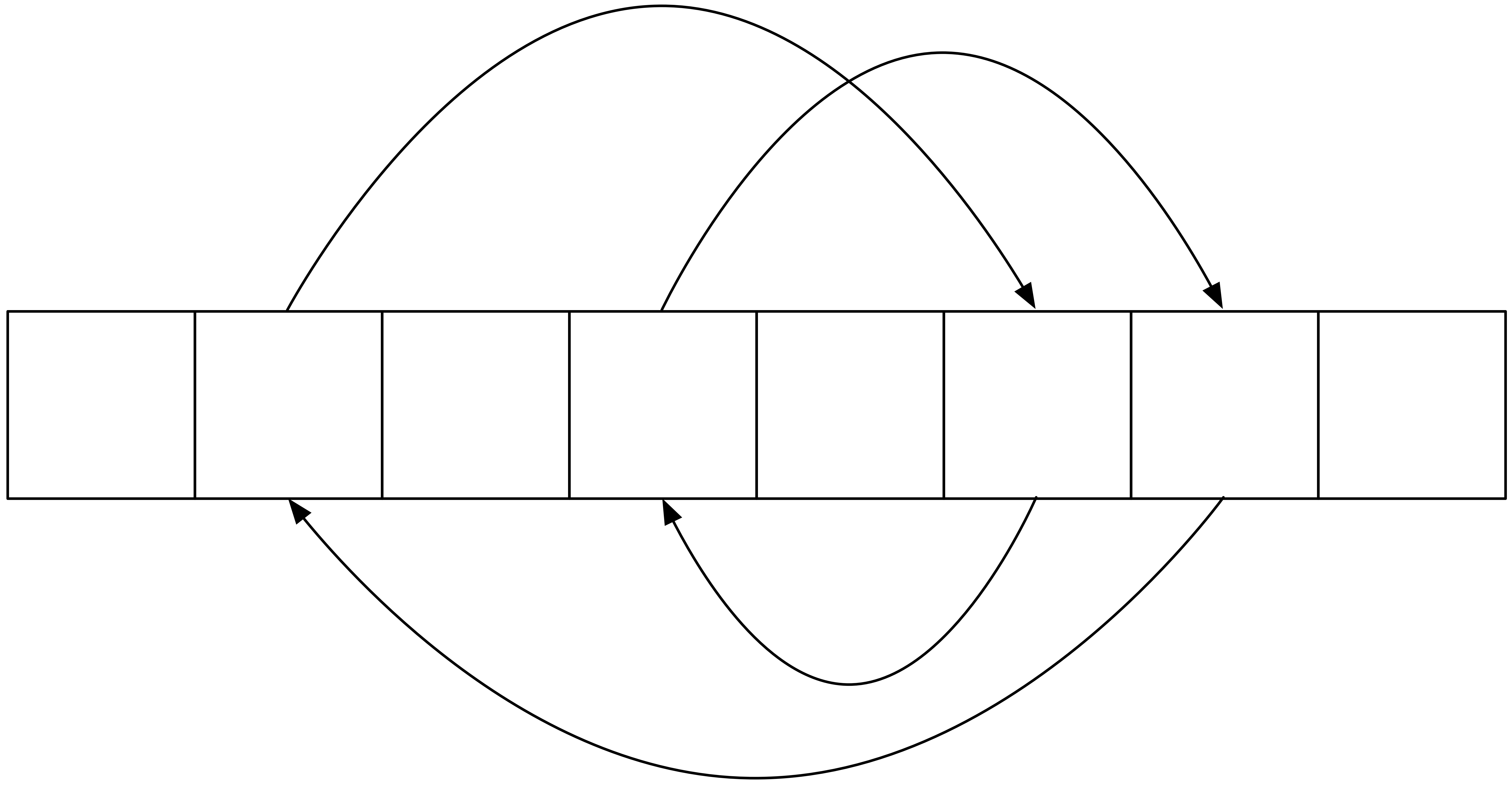
- Example implementation of new operation:

```
clone_ptr clone() const { return make_clone<T>(**this); }
```

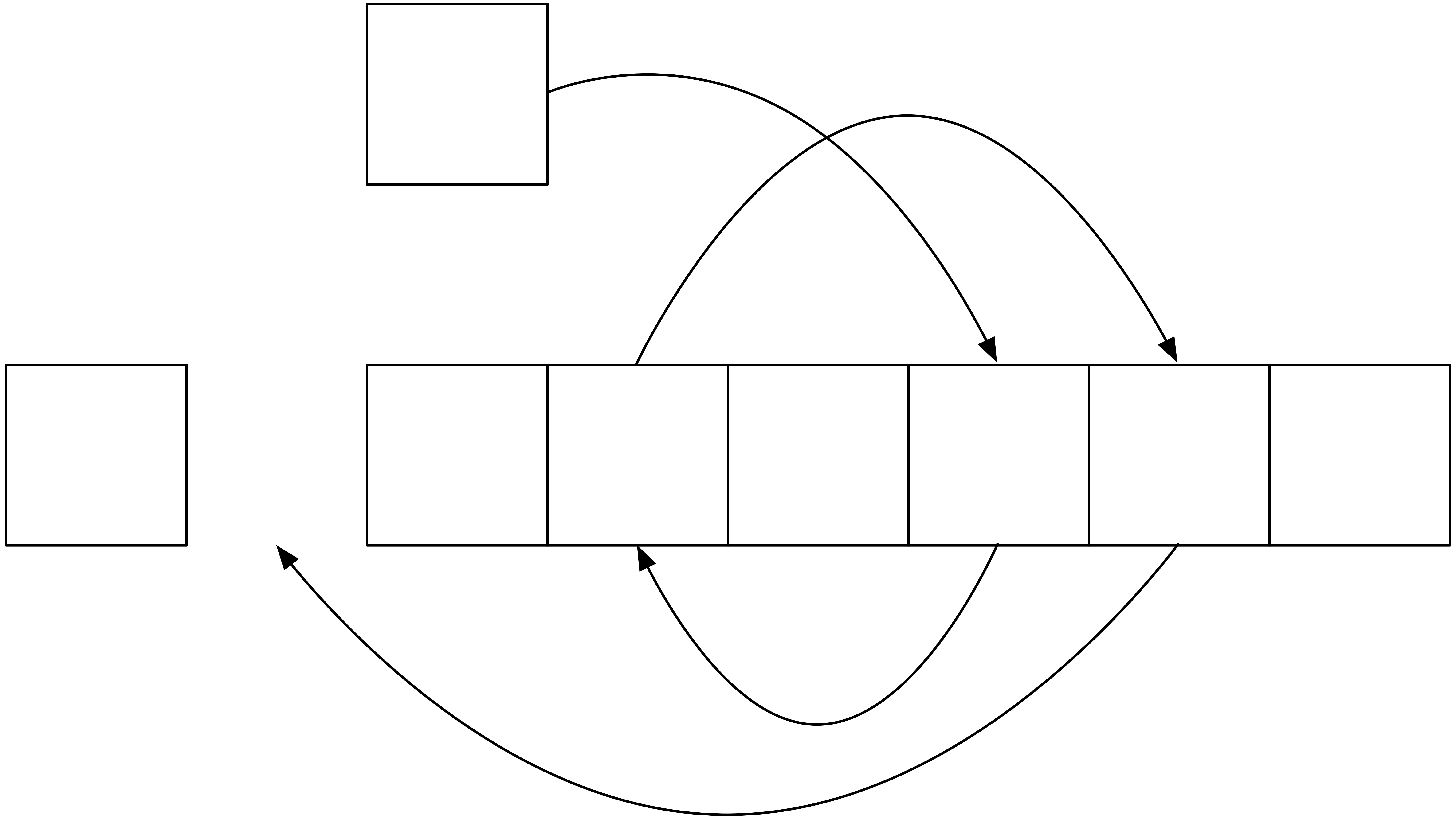
“What’s in the box?”

– Seven

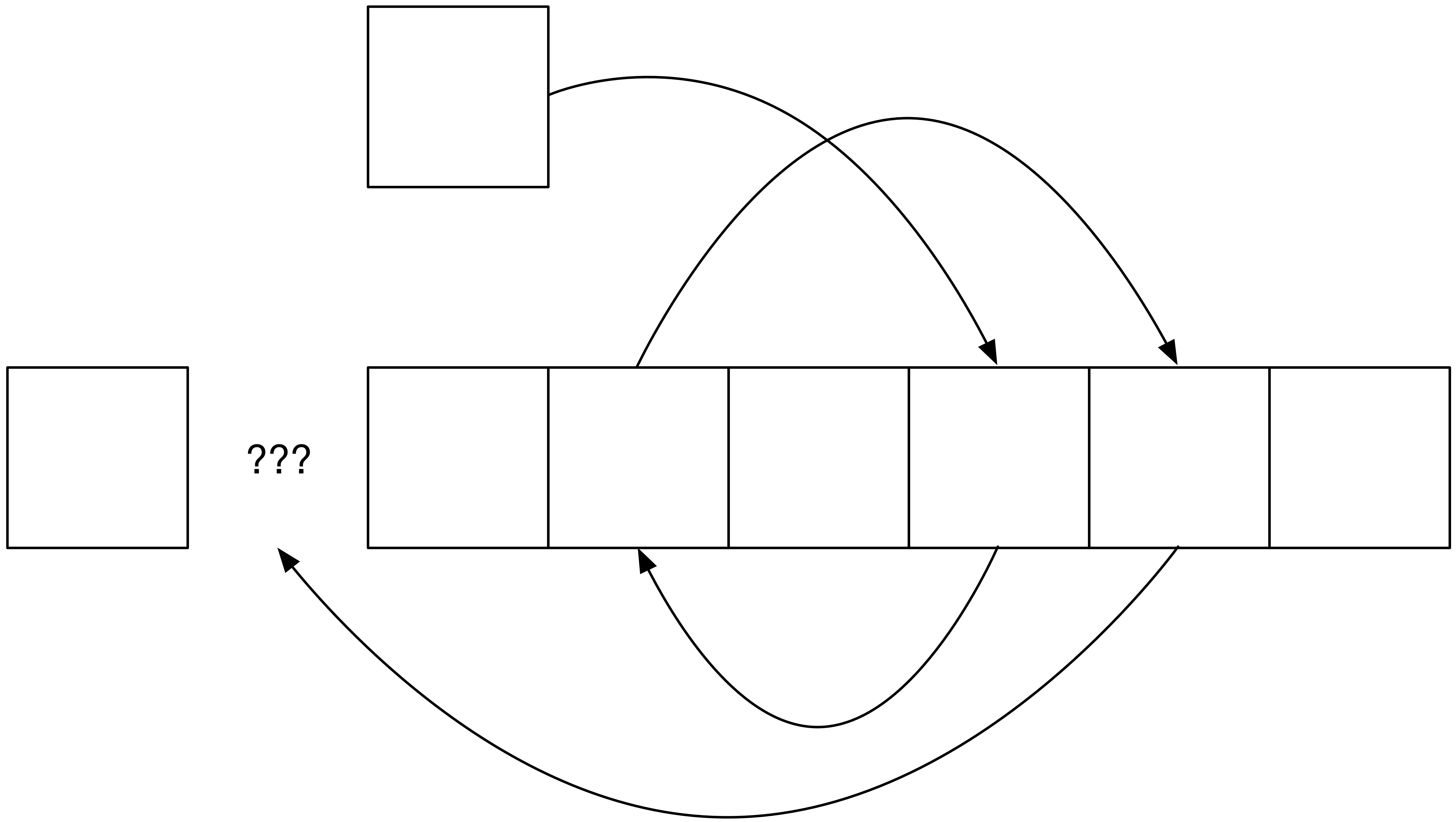
The Permutation Paradox



The Permutation Paradox

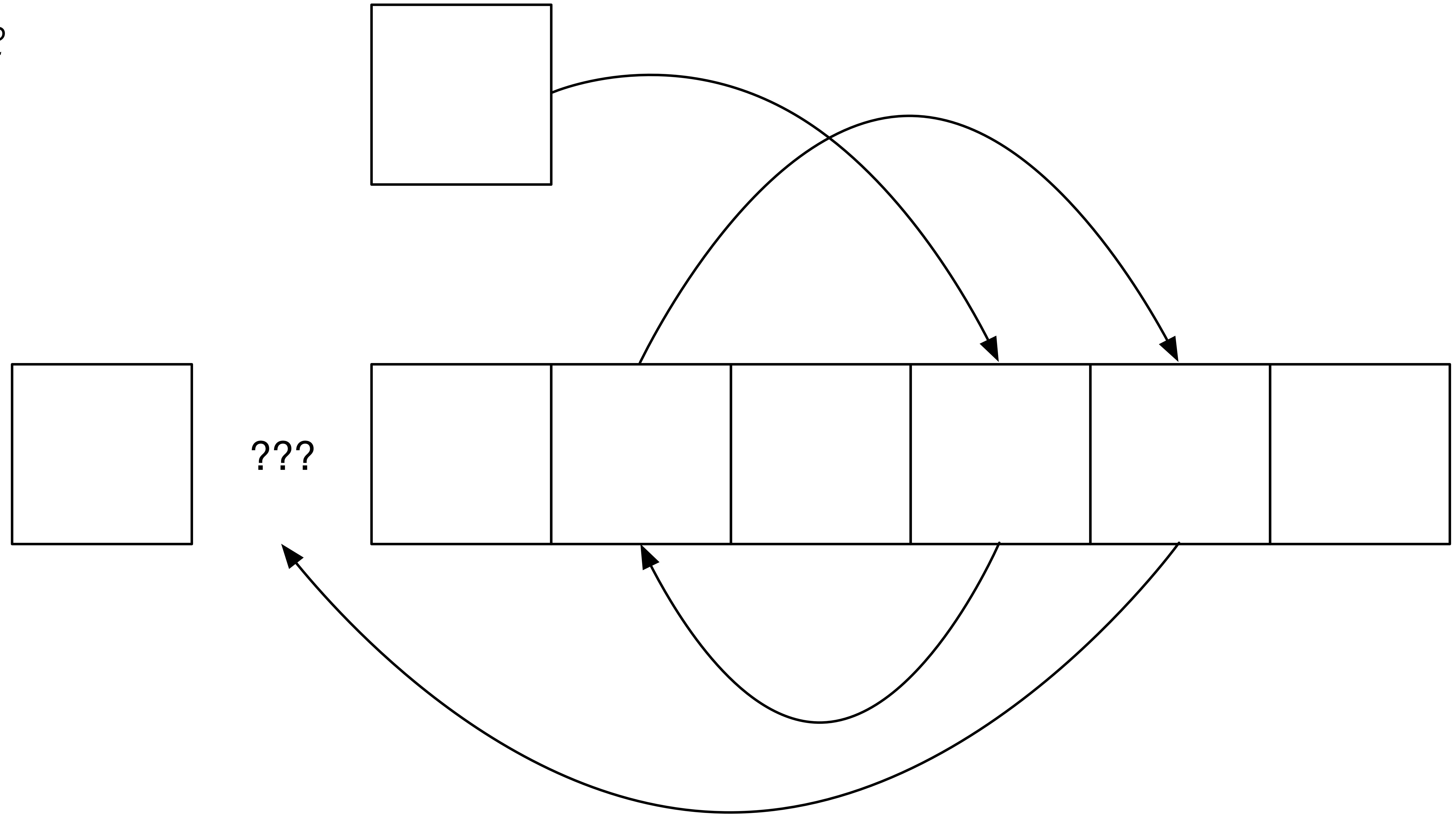


The Permutation Paradox



The Permutation Paradox

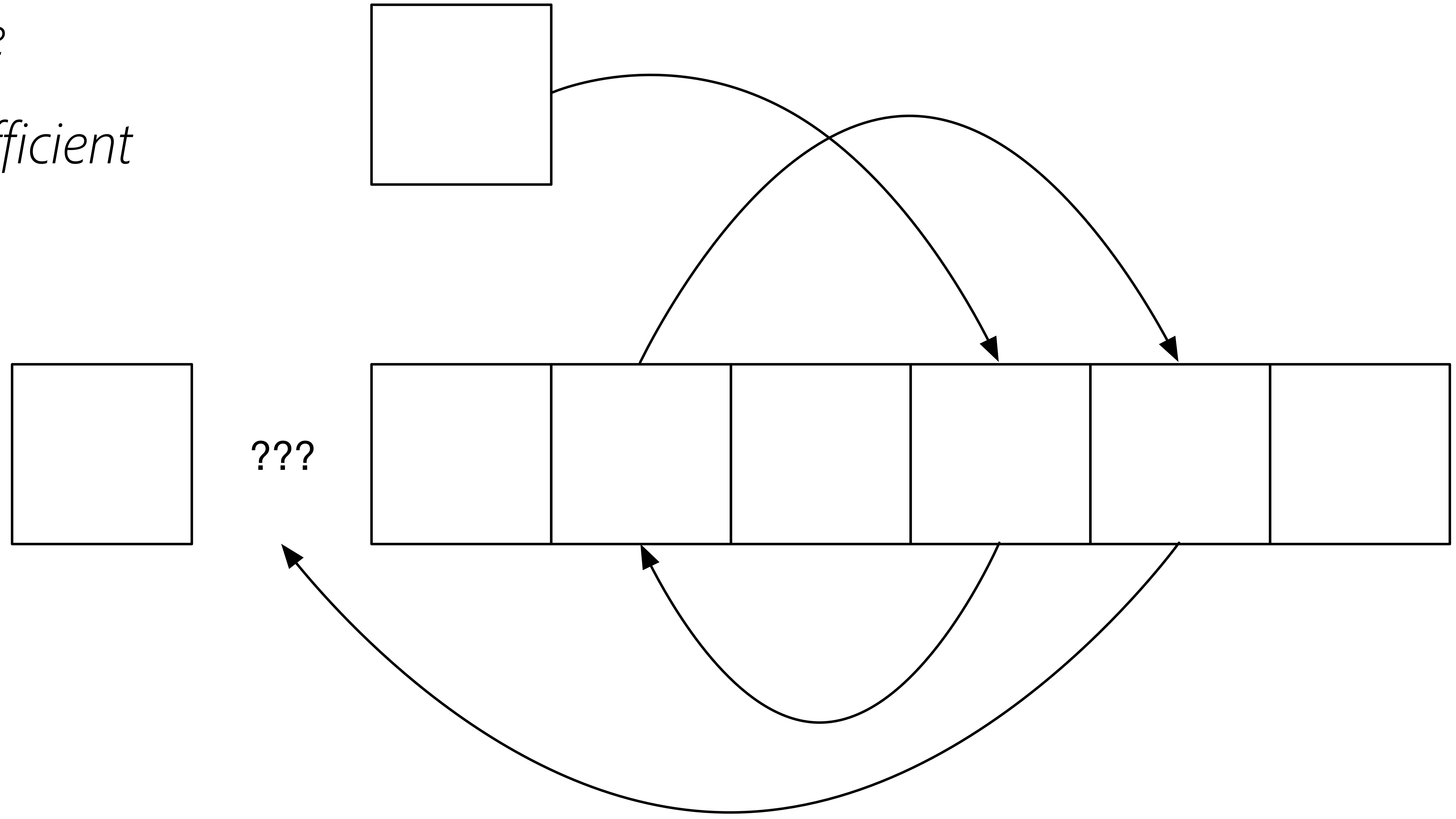
nothing ⇒ unsafe



The Permutation Paradox

nothing ⇒ unsafe

something ⇒ inefficient



The Permutation Paradox

The Permutation Paradox

"There is a duality between transformations and the corresponding actions: An action is definable in terms of a transformation and vice versa:

The Permutation Paradox

"There is a duality between transformations and the corresponding actions: An action is definable in terms of a transformation and vice versa:

```
void a(T& x) { x = f(x); } // action from transformation
```

and

```
T f(T x) { a(x); return x; } // transformation from action
```

The Permutation Paradox

"There is a duality between transformations and the corresponding actions: An action is definable in terms of a transformation and vice versa:

```
void a(T& x) { x = f(x); } // action from transformation
```

and

```
T f(T x) { a(x); return x; } // transformation from action
```

Despite this duality, independent implementations are sometimes more efficient, in which case both action and transformation need to be provided."

– *Elements of Programming (section 2.5)*

“It’s not that I’m lazy, it’s that I just don’t care.”

– Office Space

Good Code

Good code is *correct*

Consistent; without contradiction

Good code has *meaning*

Correspondence to an entity; specified, defined

Good Code

Good code is *correct*

Consistent; without contradiction

Good code has *meaning*

Correspondence to an entity; specified, defined

Good code is *efficient*

Good Code

Good code is *correct*

Consistent; without contradiction

Good code has *meaning*

Correspondence to an entity; specified, defined

Good code is *efficient*

Maximum effect with minimum resources

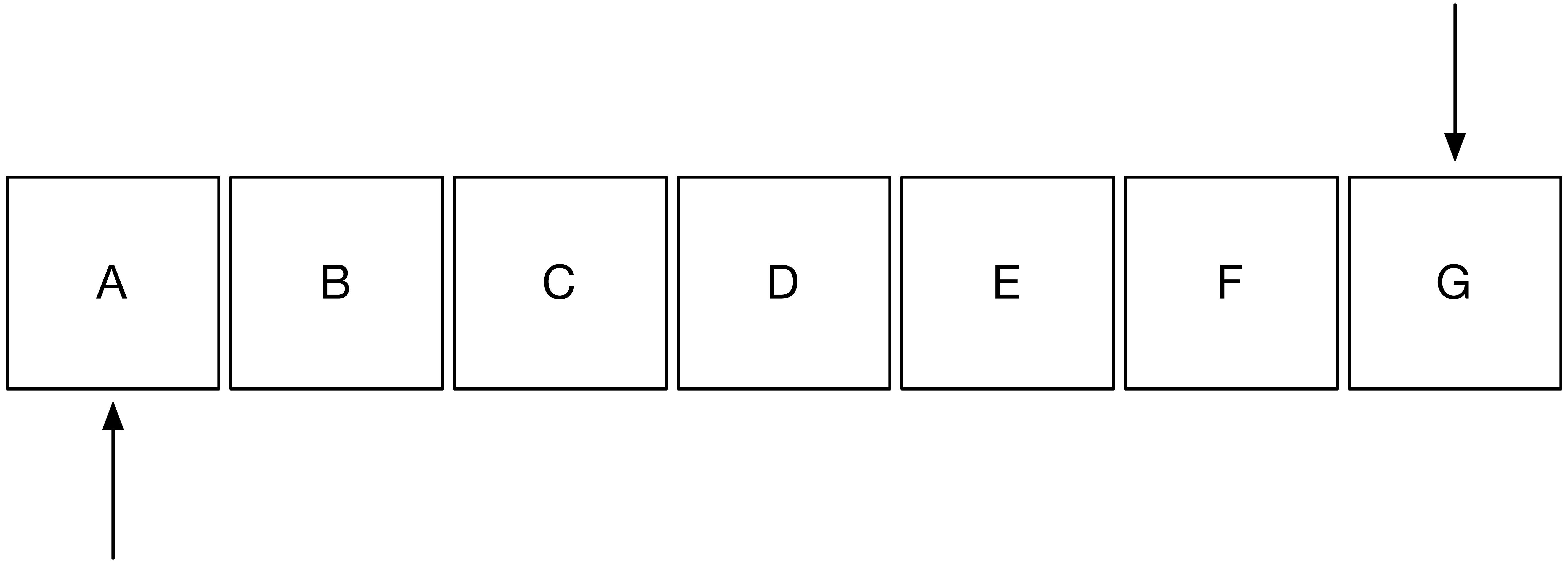
Efficiency

Efficiency

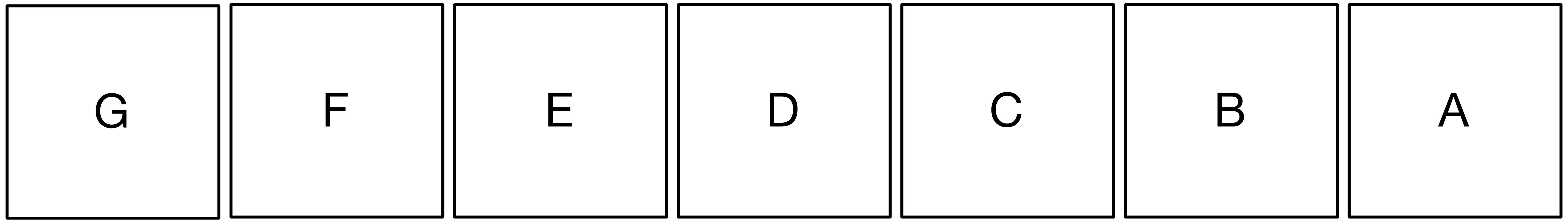
Choice of data structures and algorithms

Choice of what to optimize for

Efficiency



Efficiency



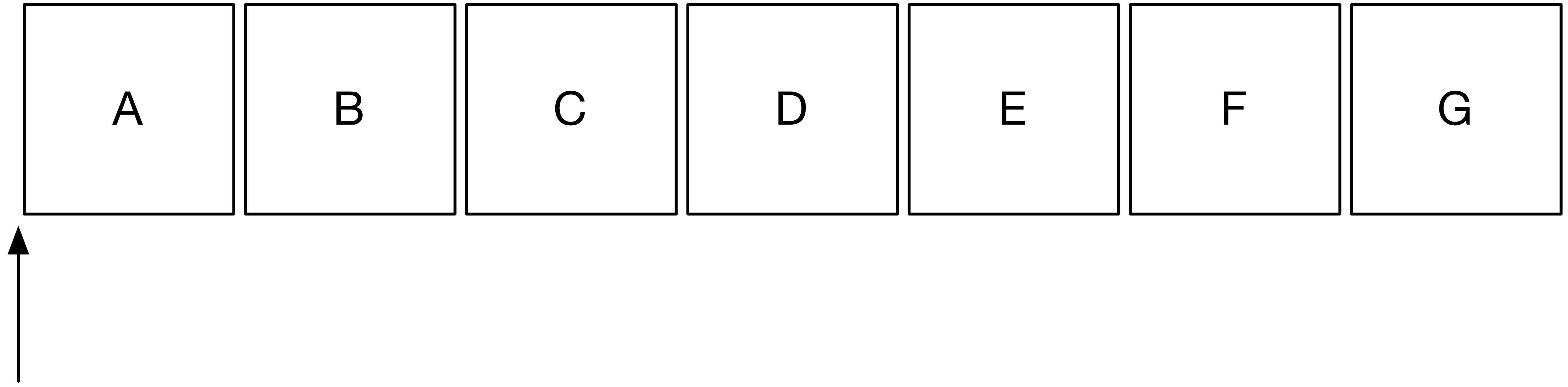
Efficiency

```
template <class ForwardIterator, class N>
auto reverse_n(ForwardIterator f, N n) {
    if (n < 2) return next(f, n);

    auto h = n / 2;
    auto m1 = reverse_n(f, h);
    auto m2 = next(m1, n % 2);
    auto l = reverse_n(m2, h);
    swap_ranges(f, m1, m2);
    return l;
}

template <class ForwardIterator>
void reverse(ForwardIterator f, ForwardIterator l) {
    reverse_n(f, distance(f, l));
}
```

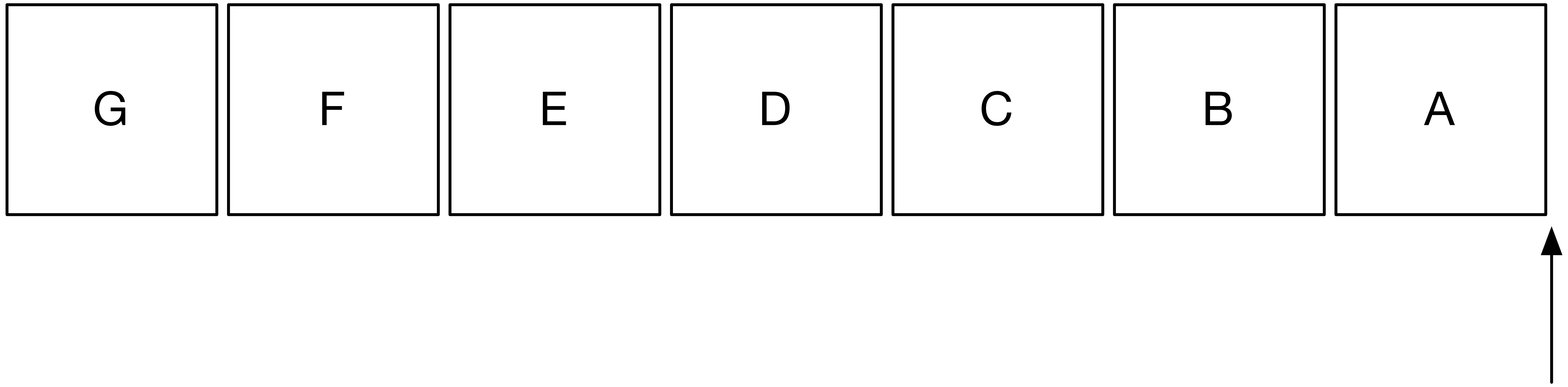
O(n log n)

Elements of Programming, 10.3

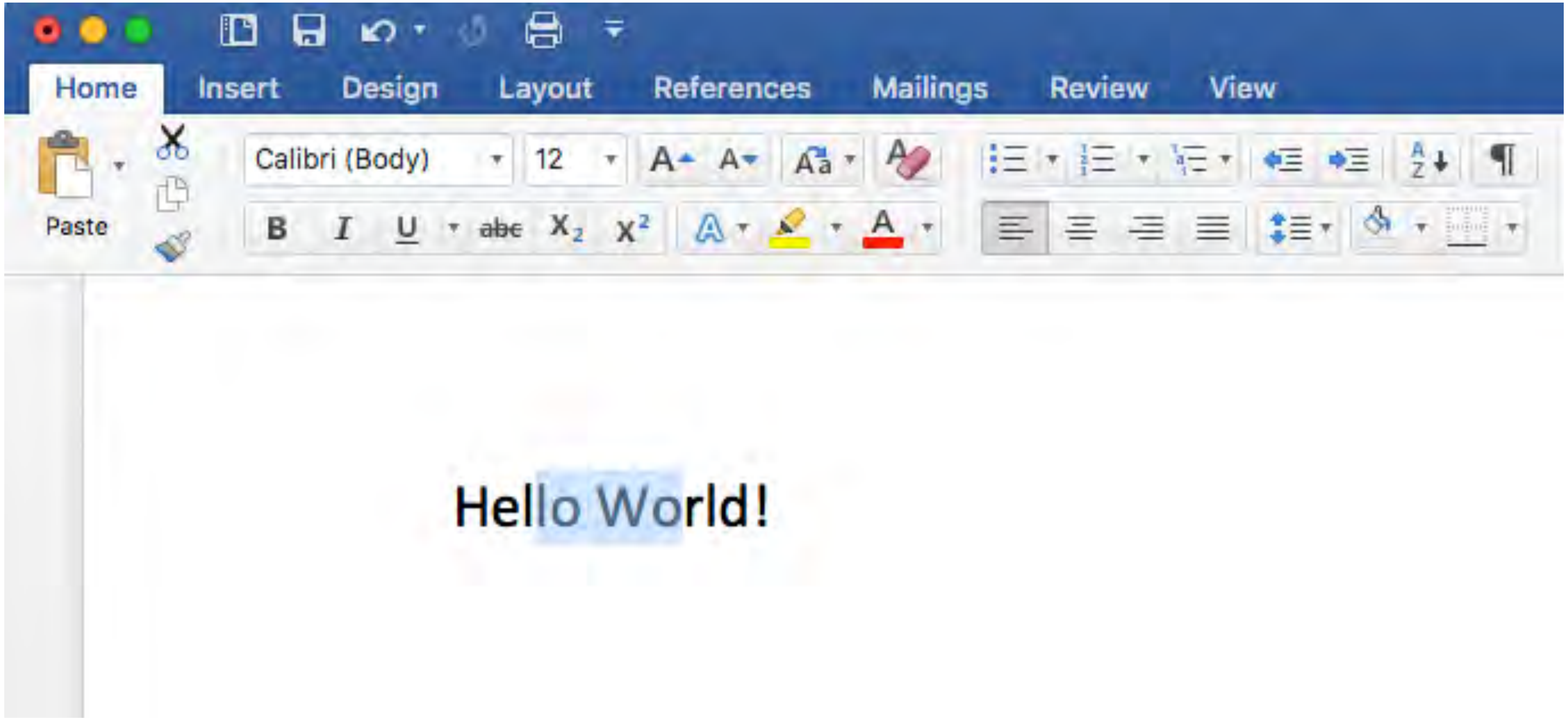
Efficiency



Efficiency

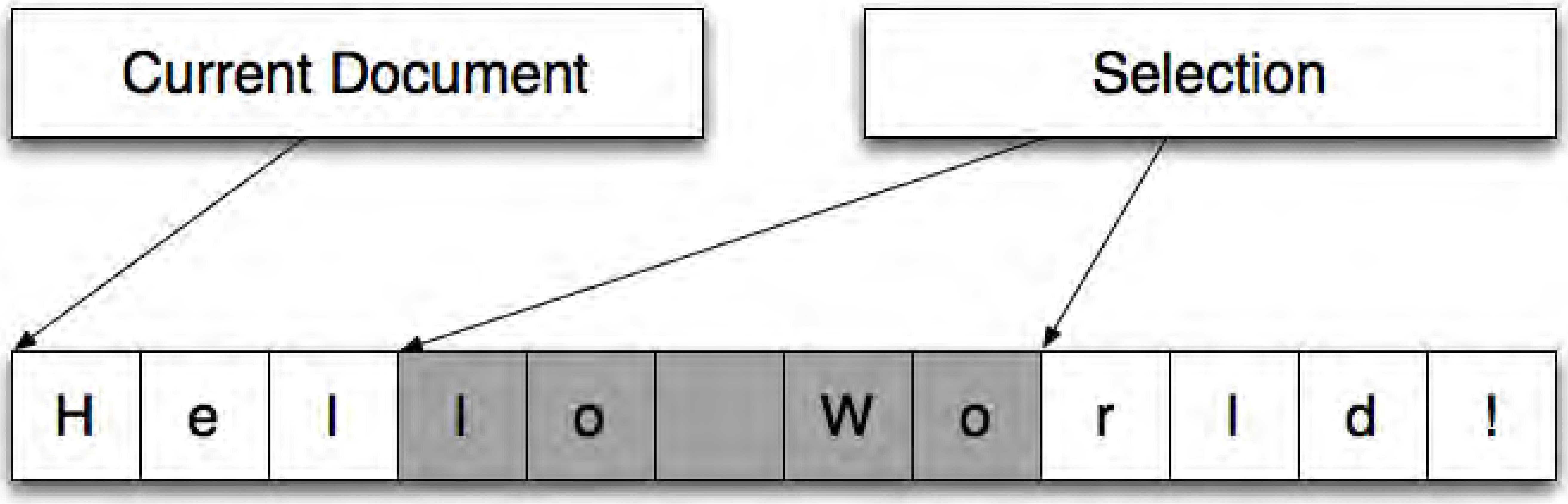


Simple Word Model



Simple Word Model

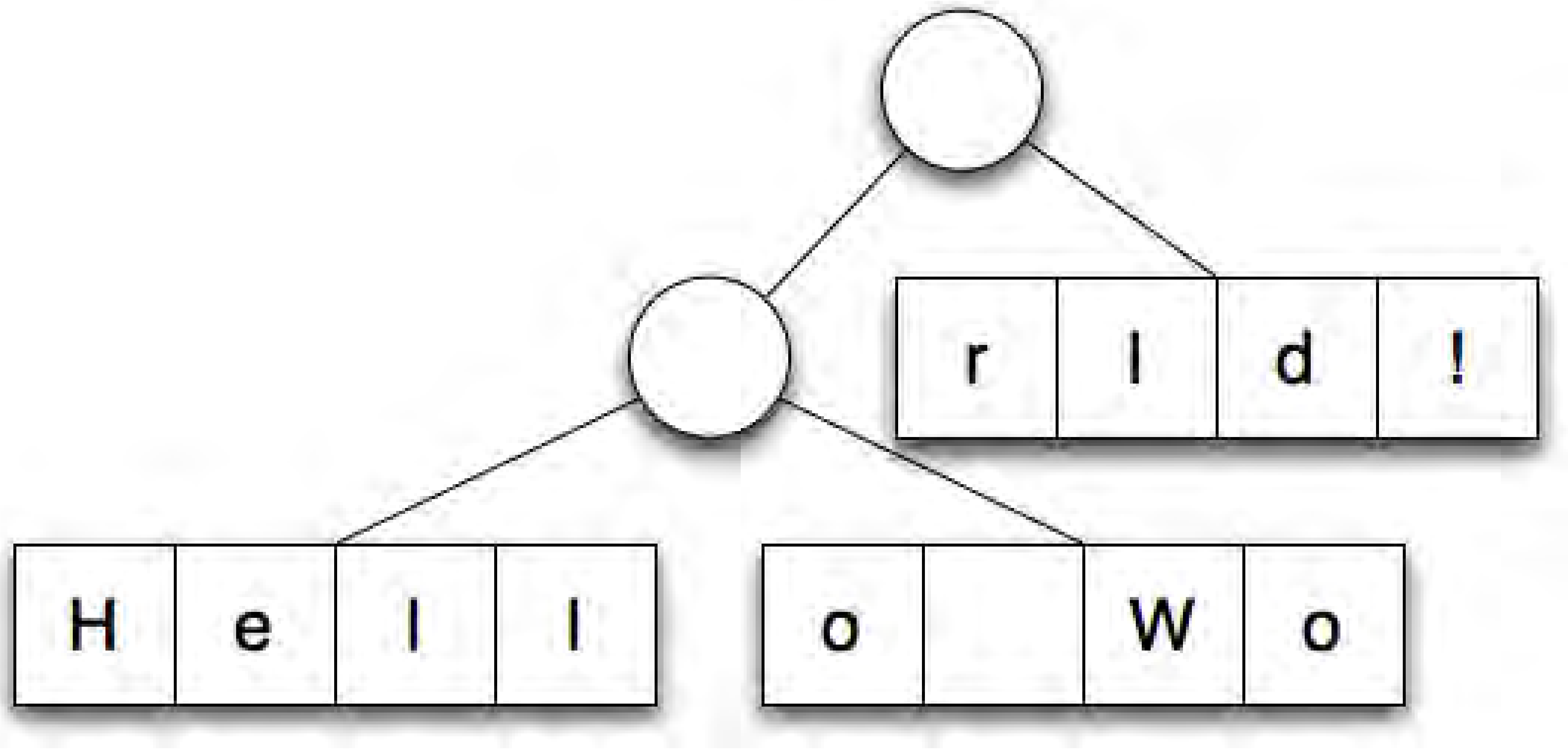
- Current Document
- Selection
  - Provides a range; an empty range denotes a location



More Complex Word Model

- Need to be able to set the selection in “constant” time
  - This would imply a vector data structure
- Also need constant time insert and erase
  - This would imply a list data structure

- Solution: a more complex data structure such as a rope



"I don't smoke, I don't drink… I recycle…"

– 50/50

Good Code

Good code is *correct*

Consistent; without contradiction

Good code has *meaning*

Correspondence to an entity; specified, defined

Good code is *efficient*

Maximum effect with minimum resources

Good code is r*eusable*

Applicable to multiple problems; general in purpose

Reusable

Reusable

Concrete but of general use, i.e. numeric algorithms, utf conversions, …

Generic when algorithm is useful with different models

  Sometimes faster to convert one model to another

Runtime dispatched when types not known at compile time

Reusable

Reusable

Minimize client dependencies and intrusive requirements

Separate data structures from algorithms

Reusable

Reusable

```
template <class T, class InputIterator, class OutputIterator>
OutputIterator copy_utf(InputIterator first, InputIterator last,
       OutputIterator result);


const char str[] = u8"Hello World!";
vector<uint16_t> out;
copy_utf<uint16_t>(begin(str), end(str), back_inserter(out));
```

"You mean we're in the future."

– Back to the Future Part II

Why Status Quo Will Fail

Why Status Quo Will Fail

"I've assigned this problem [binary search] in courses at Bell Labs and IBM. Professional programmers had a *couple of hours* to convert the description into a programming language of their choice; a high-level pseudo code was fine… Ninety percent of the programmers found bugs in their programs (and I wasn't always convinced of the correctness of the code in which no bugs were found)."

– Jon Bentley, Programming Pearls, 1986

Why Status Quo Will Fail

```
int* lower_bound(int* first, int* last, int value)
{
    while (first != last)
    {
        int* middle = first + (last - first) / 2;

        if (*middle < value) first = middle + 1;
        else last = middle;
    }

    return first;
}
```

Signs of Hope

Elements of Programming

Concepts aren't dead yet in C++

Increased interest in new languages and formalisms

Renewed interest in Communication Sequential Processes

Renewed interest in Functional Programming ideas

Rise of Reactive Programming & Functional Reactive Programming

Work Continues

Work Continues

Generating Reactive Programs for Graphical User Interfaces from Multi-way Dataflow Constraint Systems, GPCE 2015, Gabriel Foust, Jaakko Järvi, Sean Parent

One Way To Select Many, ECOOP 2016, Jaakko Järvi, Sean Parent

http://sean-parent.stlab.cc/papers-and-presentations

https://github.com/stlab

Write Better Code

Adobe