Static Inlining

Notes on *Secrets of the Glasgow Haskell Compiler Inliner*
(Simon Peyton Jones and Simon Marlow, 1999)

Emily Sillars

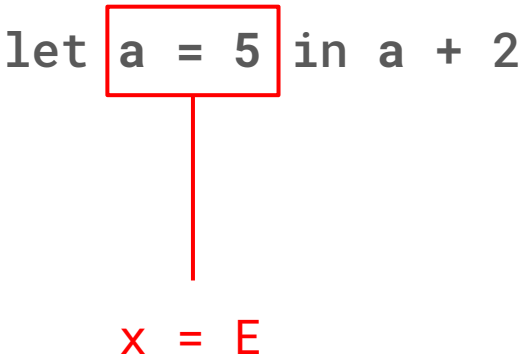
What is inlining?

Given x = E, at one place where there is an x,
replace x with E

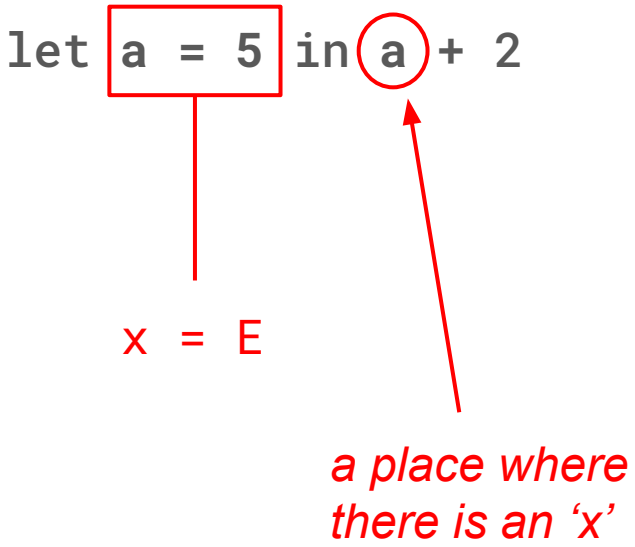
Given x = E, *at one place where there is an x*, **replace x with E**

```
let a = 5 in a + 2
```

Given x = E, *at one place where there is an x*, **replace x with E**



Given x = E, *at one place where there is an x*, **replace x with E**



Given x = E, *at one place where there is an x*, **replace x with E**

```
let a = 5 in a + 2
```

Given x = E, *at one place where there is an x*, **replace x with E**

```
let a = 5 in (5) + 2
```

3 kinds of inlining…

1) Inlining Itself
2) Dead Code Elimination
3) β - reduction (beta reduction)

3 kinds of inlining…

1) Inlining Itself *HARD*
2) Dead Code Elimination *EASY*
3) β - reduction (beta reduction) *EASY*

1) Inlining Itself

let - binding replace LHS with RHS

```
let { f = \x -> x*3 } in f (a + b) - c
```

⇒ *inline f*

```
let { f = \x -> x*3 } in (\x -> x*3) (a + b) - c
```

2) Dead Code Elimination

Delete binding if no longer used

```
let { f = \x -> x*3 } in (\x -> x*3) (a + b) - c
```

⇒ *dead f*

```
(\x -> x*3) (a + b) - c
```

f not present anywhere!

3) β - reduction

Turn **lambda application** into let-binding

(lambda) *(argument)*

```
(\x -> x*3) (a + b) - c
```

⇒ *β (beta)*

```
(let { x = a+b } in x*3) - c
```

A quick example …

```
let { f = \x -> x*3 } in f (a + b) - c
```

⇒ *inline f*

```
let { f = \x -> x*3 } in (\x -> x*3) (a + b) - c
```

⇒ *dead f*

```
(\x -> x*3) (a + b) - c
```

⇒ *β (beta)*

```
let {x = (a + b) } in x*3 - c
```

⇒ *inline x*

```
let {x = (a + b) } in (a + b)*3 - c
```

⇒ *dead x*

```
(a + b)*3 - c
```

```
let { f = \x -> x*3 } in f (a + b) - c
```

⇒ *inline f*

```
let { f = \x -> x*3 } in (\x -> x*3) (a + b) - c
```

⇒ *dead f*

```
(\x -> x*3) (a + b) - c
```

⇒ *β (beta)*

```
let {x = (a + b) } in x*3 - c
```

⇒ *inline x*

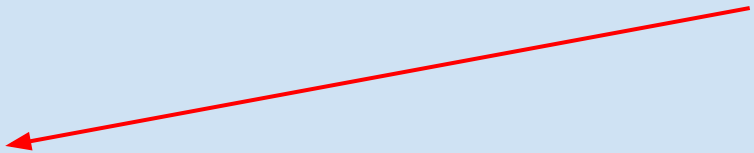
```
let {x = (a + b) } in (a + b)*3 - c
```

⇒ *dead x*

```
(a + b)*3 - c
```

Hey, Look!

I don't have to store a function with its environment now!



To inline, or not to inline?

To inline, or not to inline?

1) Inlining Itself *HARD*
2) Dead Code Elimination *EASY*
3) β - reduction (beta reduction) *EASY*

To inline, or not to inline?

1) Inlining Itself *HARD*
2) ~~Dead Code Elimination~~ *EASY*
3) β - reduction (beta reduction) *EASY*

Name - Capture Problem: How to inline C?

```
let c =  a + b in

   let a = 7 in

      c+a
```

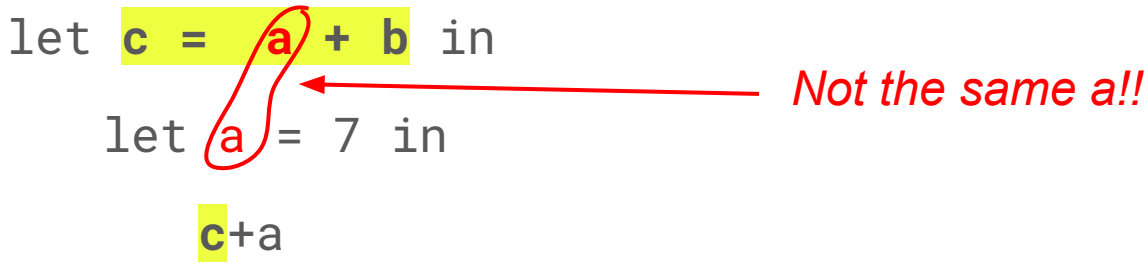
Name - Capture Problem: How to inline C?

```
let c =  a + b in
    let a = 7 in
        c+a
```

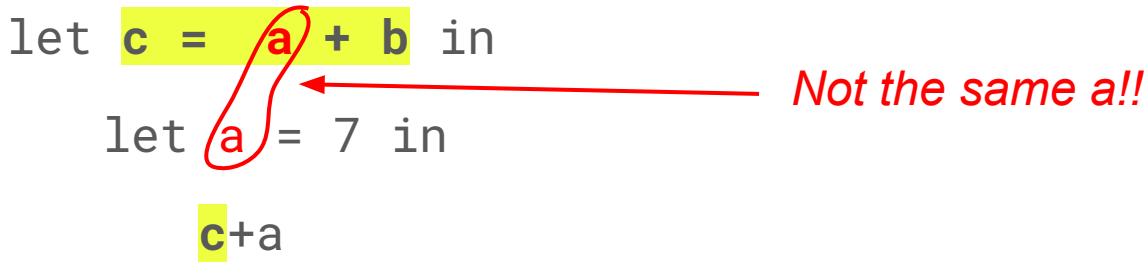
Name - Capture Problem: How to inline C?

```
let c =  a + b in

    let a = 7 in

        c+a
```

Name - Capture Problem: How to inline C?



Name - Capture Problem: How to inline C?



Solution: rename bound variable

Name - Capture Solution: How to inline C?

```
let c =  a + b in
    let s796 = 7 in
        c+s796
```

Name - Capture Solution: How to inline C?

```
let c =  a + b in
    let s796 = 7 in
        (a + b)+s796
```

General Rule to Replace x with E in M

Case 1) subst (λx.M) [E/x] = λx.M

Case 2) subst (λw.M) [E/x] = λw.(subst M [E/x])

if w does not occur free in E

Case 3) Otherwise, need to rename w to get a case 2.

Note: *subst M [E/x]* means “the result of replacing x with E in M”

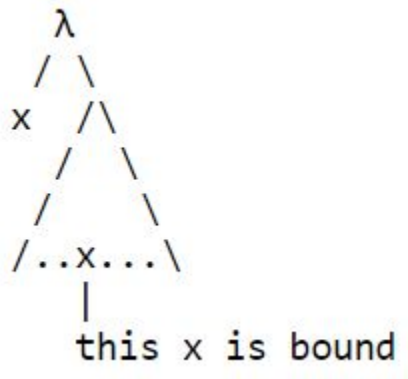
Note: In the rule above, M is a lambda abstraction

Back up, let's talk about λ Calculus!!

"**Bound Variable**: a variable that is associated with some lambda.

Free Variable: a var that is *not* associated with any lambda.

Intuitively, in lambda-expression M, variable x is bound if, in the abstract-syntax tree, x is in the subtree of a lambda with left child x"



(Horwitz, "Lambda Calculus (Part I)")

1) *subst (λx.M) [E/x] = λx.M*
2) *subst (λw.M) [E/x] = λw.(subst M [E/x])*

if w does not occur free in E

3) *Otherwise, need to rename w to get a case 2.*

Goal: Inline c in M

```
let c =  a + b in
    let a = 7 in
    c+a
```

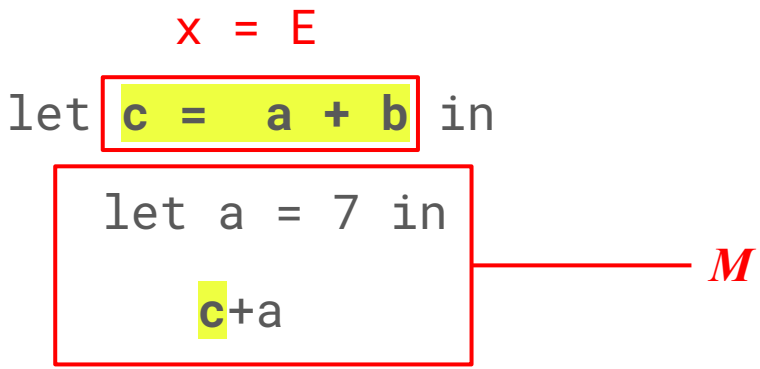
1) *subst (λx.M) [E/x] = λx.M*
2) *subst (λw.M) [E/x] = λw.(subst M [E/x])*

if w does not occur free in E

3) *Otherwise, need to rename w to get a case 2.*

Goal: Inline c in M

subst M [E/c]



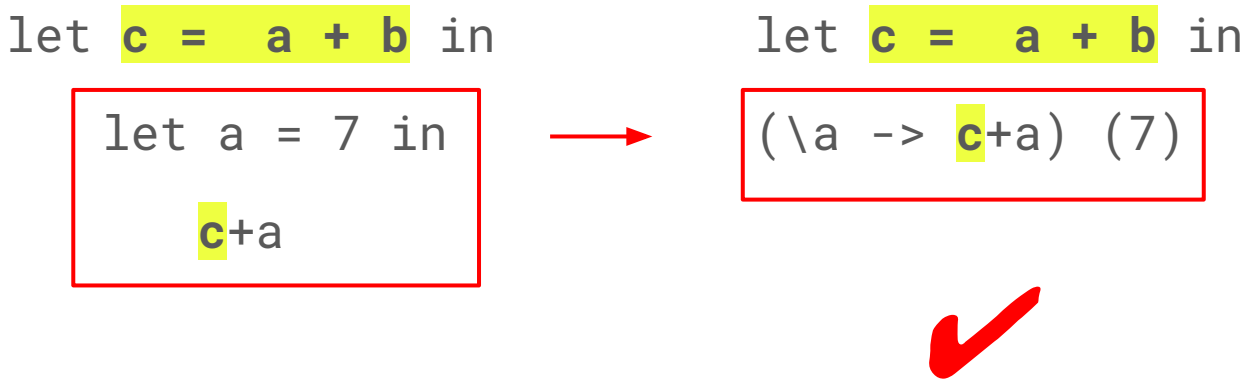
We need M in lambda form to use the substitution rule!

Go the other way… β - expansion?

```
let c =  a + b in
    let a = 7 in
        c+a
```

```
let c =  a + b in
(\a -> c+a) (7)
```

Go the other way… β - expansion?



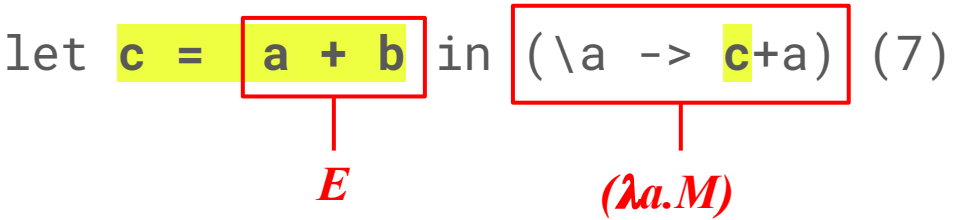
1) *subst (λx.M) [E/x] = λx.M*
2) *subst (λw.M) [E/x] = λw.(subst M [E/x])*

if w does not occur free in E

3) *Otherwise, need to rename w to get a case 2.*

Goal: Inline c in M

subst (λa.M) [E/c]



Just like how **w** and **x** are not the same, neither are **a** and **c,** so this is case 2 or 3!

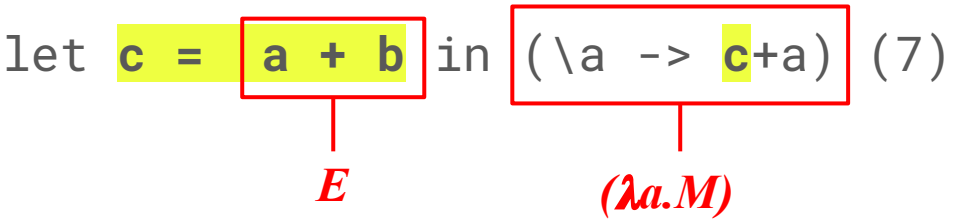
1) *subst (λx.M) [E/x] = λx.M*
2) *subst (λw.M) [E/x] = λw.(subst M [E/x])*

if w does not occur free in E

3) *Otherwise, need to rename w to get a case 2.*

Goal: Inline c in M

subst (λa.M) [E/c]



Q: Does **a** not occur free in **E**?

A: Well, no, **a** is not associated with some lambda in **a + b**, so **a** occurs free in **E***. Therefore, case 3!

*Somewhere outside this "`let c = a + b`" expression, a and b must be bound for this example to make sense at all; but just looking at the expression a + b, both a and b are free.

1) *subst (λx.M) [E/x] = λx.M*
2) *subst (λw.M) [E/x] = λw.(subst M [E/x])*

if w does not occur free in E

3) *Otherwise, need to rename w to get a case 2.*

Goal: Inline c in M

subst (λs796.M) [E/c]

```
let c =  a + b in (\s796 -> c+s796) (7)
```

Case 3, so let's rename a to s796…

1) *subst (λx.M) [E/x] = λx.M*
2) *subst (λw.M) [E/x] = λw.(subst M [E/x])*

if w does not occur free in E

3) *Otherwise, need to rename w to get a case 2.*

Goal: Inline c in M

subst (λs796.M) [E/c]

```
let c = a + b in (\s796 -> c+s796) (7)
```

E ***(λs796.M)***

Q: Does **s796** not occur free in **E**?

A: **E = a + b**, so **s796** doesn't occur in E at all. So yes, does not occur free in E.

1) *subst (λx.M) [E/x] = λx.M*
2) *subst (λw.M) [E/x] = λw.(subst M [E/x])*

if w does not occur free in E

3) *Otherwise, need to rename w to get a case 2.*

Goal: Inline c in M

subst (λs796.M) [E/c]

λs796.(subst M [E/c])

```
let c =  a + b in (\s796 -> (a+b)+s796) (7)
```

1) *subst (λx.M) [E/x] = λx.M*
2) *subst (λw.M) [E/x] = λw.(subst M [E/x])*

if w does not occur free in E

3) *Otherwise, need to rename w to get a case 2.*

Goal: Inline c in M

subst (λs796.M) [E/c]

↓

λs796.(subst M [E/c])

```
let c =  a + b in (\s796 -> a+b+s796) (7)
```

⇒ *dead c*

```
(\s796 -> a+b+s796) (7)
```

⇒ *β (beta)*

```
let s796 = 7 in (a+b) + s796
```

Another Problem: Recursive bindings…

```
recursiveGroup :: Int
recursiveGroup = f
    where
     f = (g 1)
     g = h . p
     h = \y -> if y < 15 then 5 else f
     q = g
     p = \x -> if x < 10 then 10 else q (x - 1)
```

Let’s inline f…

```
recursiveGroup :: Int
recursiveGroup = f
    where
     f = (g 1)
     g = h . p
     h = \y -> if y < 15 then 5 else (g 1)
     q = g
     p = \x -> if x < 10 then 10 else q (x - 1)
```

Remove dead f…

```
recursiveGroup :: Int
recursiveGroup = f
    where

     g = h . p
     h = \y -> if y < 15 then 5 else (g 1)
     q = g
     p = \x -> if x < 10 then 10 else q (x - 1)
```

Let’s inline g…

```
recursiveGroup :: Int
recursiveGroup = f
    where

      g = h . p
      h = \y -> if y < 15 then 5 else ((h . p) 1)
      q = (h . p)
      p = \x -> if x < 10 then 10 else q (x - 1)
```

Remove dead g…

```
recursiveGroup :: Int
recursiveGroup = f
    where


     h = \y -> if y < 15 then 5 else ((h . p) 1)
     q = (h . p)
     p = \x -> if x < 10 then 10 else q (x - 1)
```

Let’s inline h…

```
recursiveGroup :: Int
recursiveGroup = f
    where
```

```
    h = \y -> if y < 15 then 5 else ((h . p) 1)
    q = ((\y -> if y < 15 then 5 else ((h . p) 1)) . p)
    p = \x -> if x < 10 then 10 else q (x - 1)
```

Huh, there's still an h inside q. Better inline h…

```
recursiveGroup :: Int
recursiveGroup = f
    where


    h = \y -> if y < 15 then 5 else ((h . p) 1)
    q = ((\y -> if y < 15 then 5 else ((h . p) 1)) . p)
    p = \x -> if x < 10 then 10 else q (x - 1)
```

Huh, there's still an h inside q. Better inline h…

```
recursiveGroup :: Int
recursiveGroup = f
    where

      h = \y -> if y < 15 then 5 else ((h . p) 1)
      q = ((\y -> if y < 15 then 5 else (((\y -> if y < 15
then 5 else ((h . p) 1)) . p) 1)) . p)
      p = \x -> if x < 10 then 10 else q (x - 1)
```

Huh, there's still an h inside q. Better inline h…

```
recursiveGroup :: Int

recursiveGroup = f

    where

     h = \y -> if y < 15 then 5 else ((h . p) 1)

     q = ((\y -> if y < 15 then 5 else (((\y -> if y < 15
then 5 else (((\y -> if y < 15 then 5 else ((h . p) 1)) . p)
1)) . p) 1)) . p)

     p = \x -> if x < 10 then 10 else q (x - 1)
```

Huh, there's still an h inside q. Better inline h…

```
recursiveGroup :: Int

recursiveGroup = f

    where

     h = \y -> if y < 15 then 5 else ((h . p) 1)

     q = ((\y -> if y < 15 then 5 else (((\y -> if y < 15
then 5 else (((\y -> if y < 15 then 5 else (((\y -> if y <
15 then 5 else ((h . p) 1) . p) 1)) . p) 1)) . p) 1)) . p)

     p = \x -> if x < 10 then 10 else q (x - 1)
```

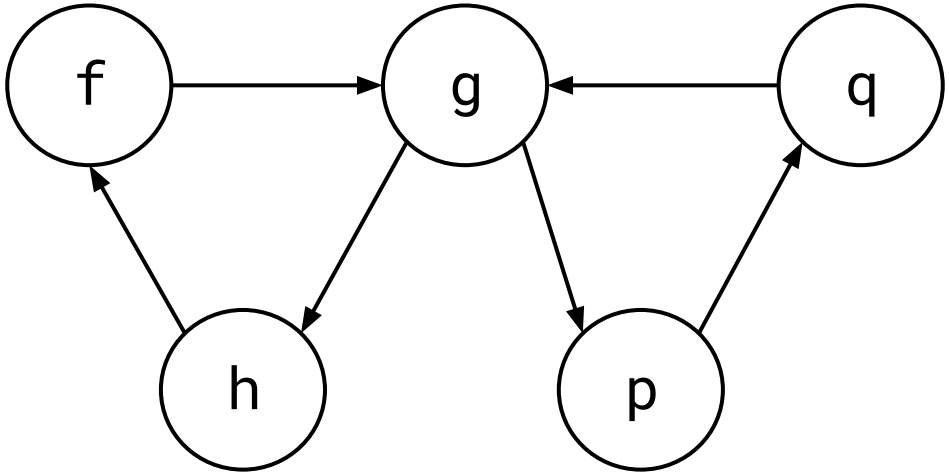
Huh, there's still an h inside q. Better inline h…

```
recursiveGroup :: Int

recursiveGroup = f

    where

     h = \y -> if y < 15 then 5 else ((h . p) 1)

     q = ((\y -> if y < 15 then 5 else (((\y -> if y < 15
then 5 else (((\y -> if y < 15 then 5 else (((\y -> if y <
15 then 5 else (((\y -> if y < 15 then 5 else ((h . p) 1) .
p) 1) . p) 1)) . p) 1)) . p) 1)) . p)

     p = \x -> if x < 10 then 10 else q (x - 1)
```

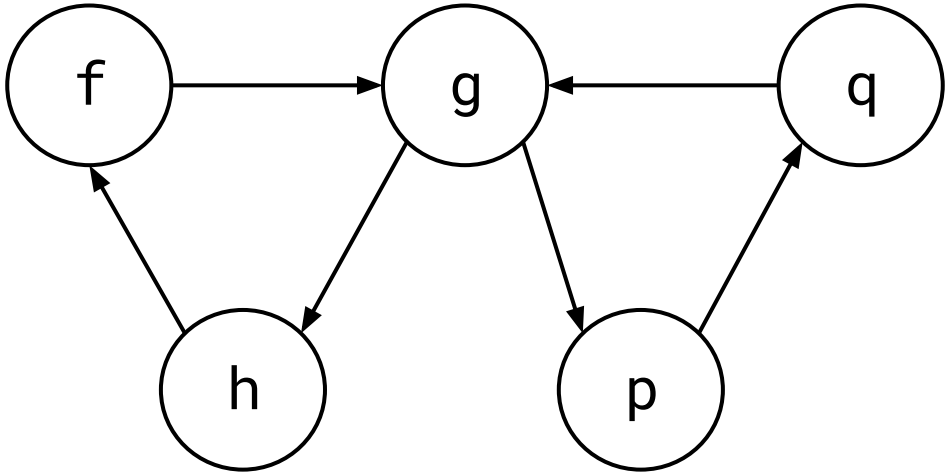
And on and on…
Will keep inlining forever!

This is bad.

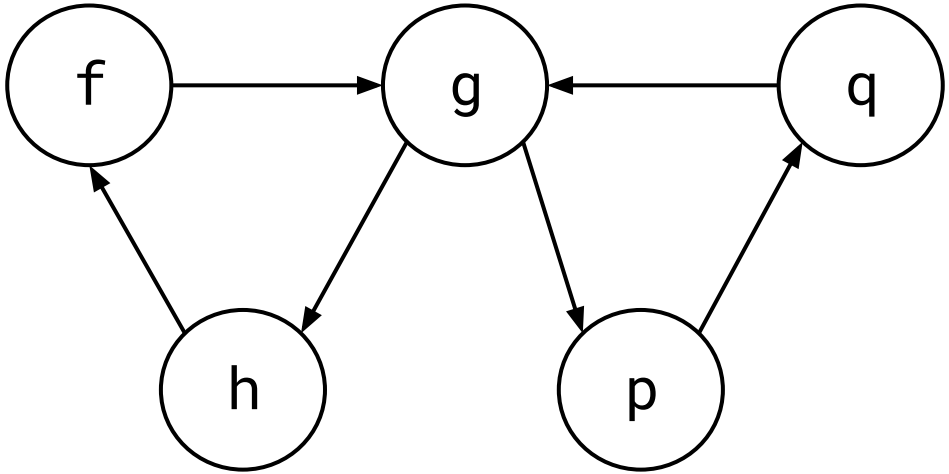
Solution: Dependency Graph, Strongly Connected Components + Loop Breakers!



1) Perform SCC on Dependency Graph

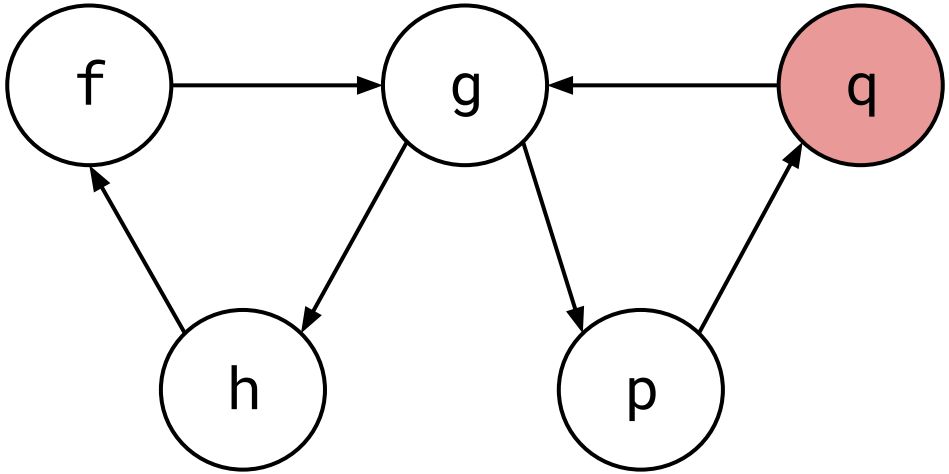


1) Perform SCC on Dependency Graph



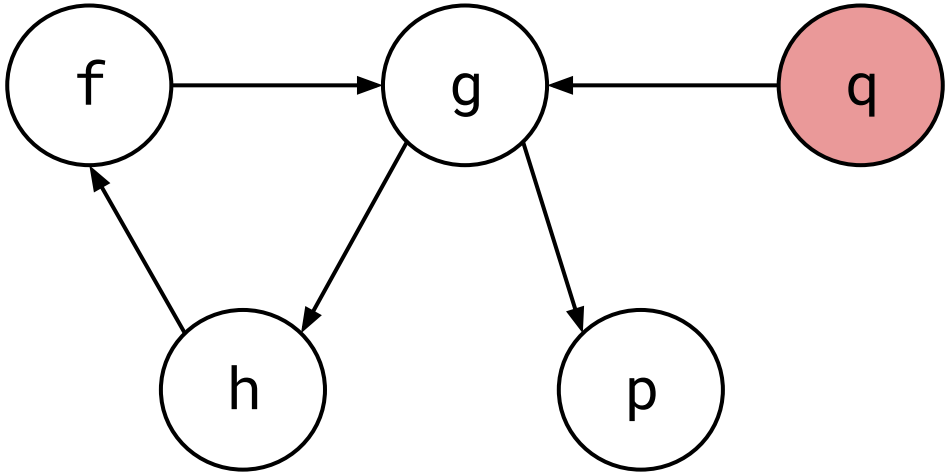
This graph is already a SCC …

2) For each SCC, pick a loop breaker based on heuristics.

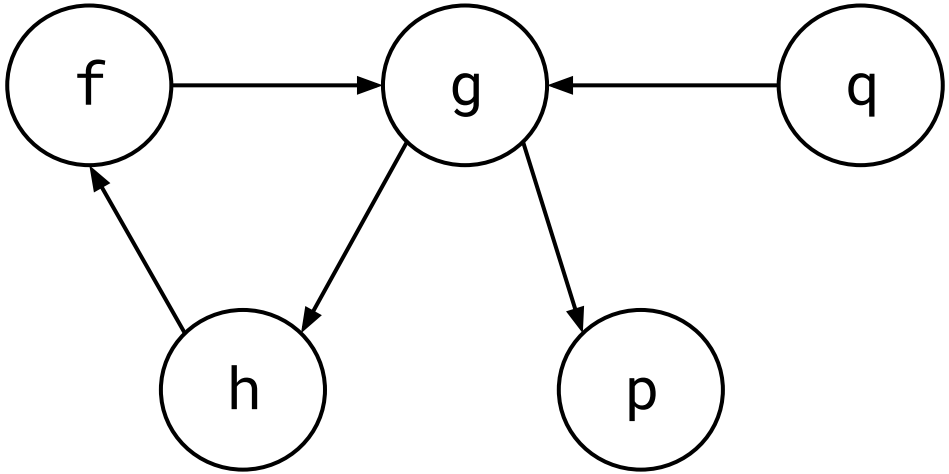


Let's arbitrarily pick q for now.

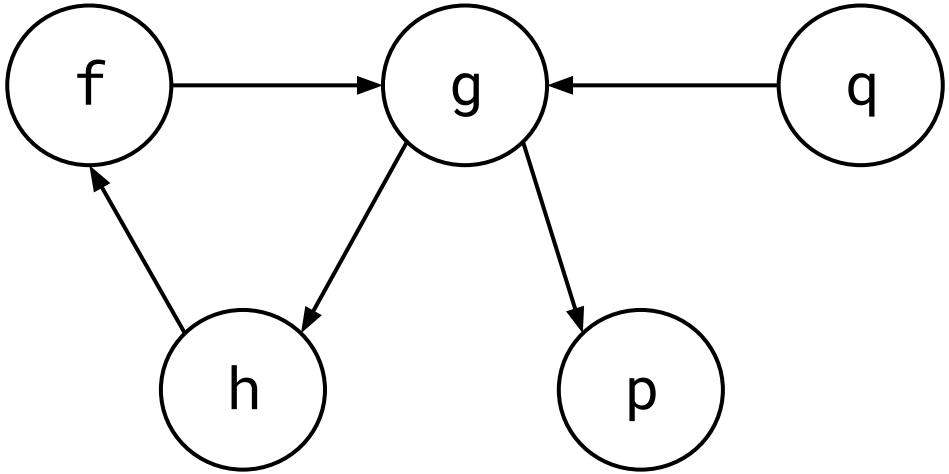
3) Delete edges pointing to loop breaker q



3) Repeat steps until no more SCC!

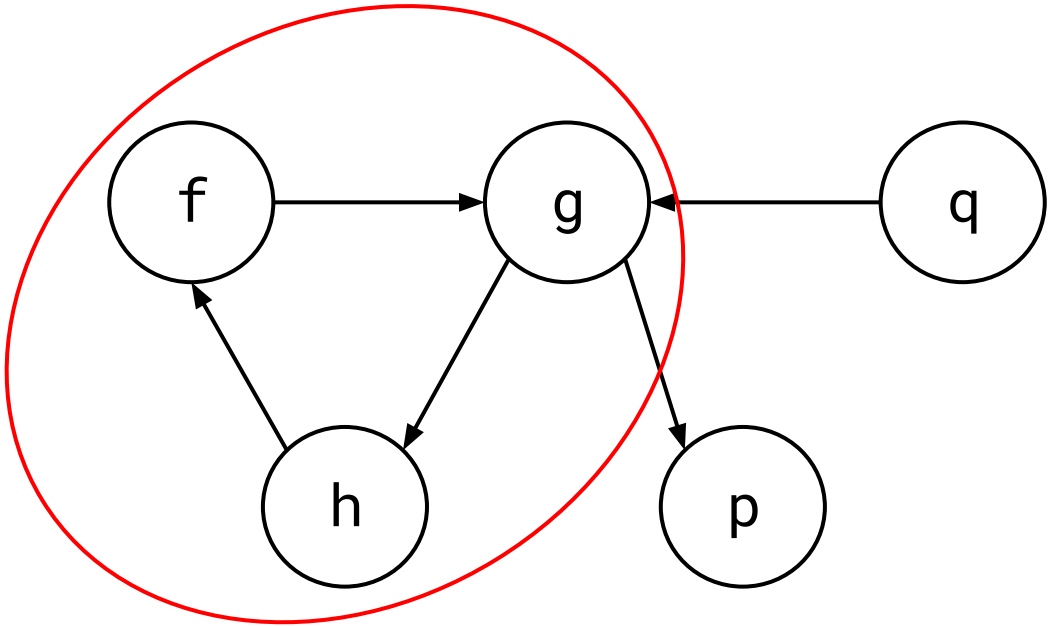


1) Perform SCC on Dependency Graph



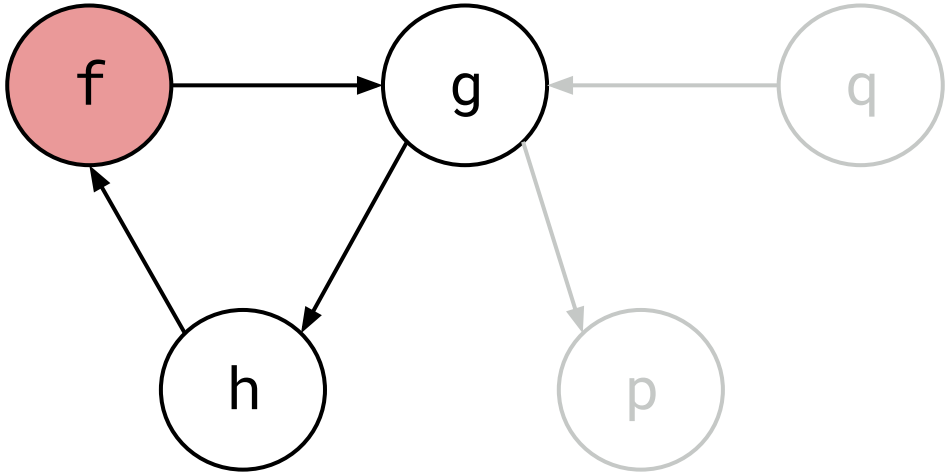
1) Perform SCC on Dependency Graph

Here's our SCC!

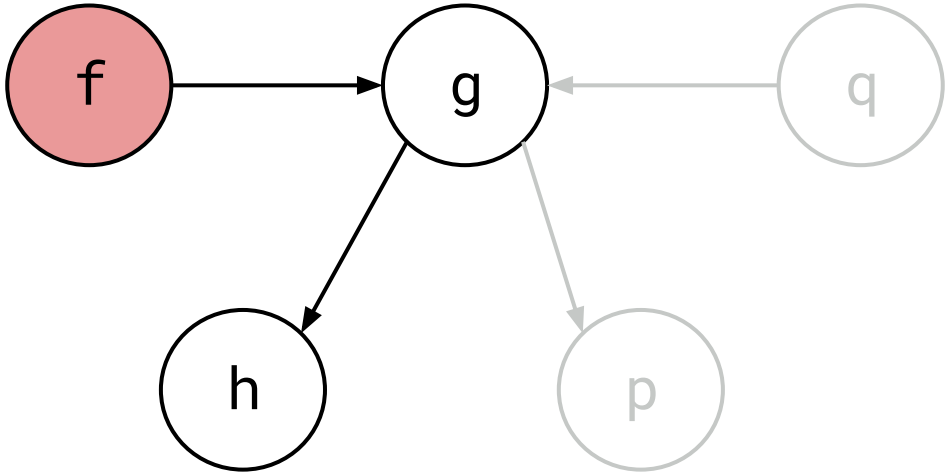


2) For each SCC, pick a loop breaker based on heuristics.

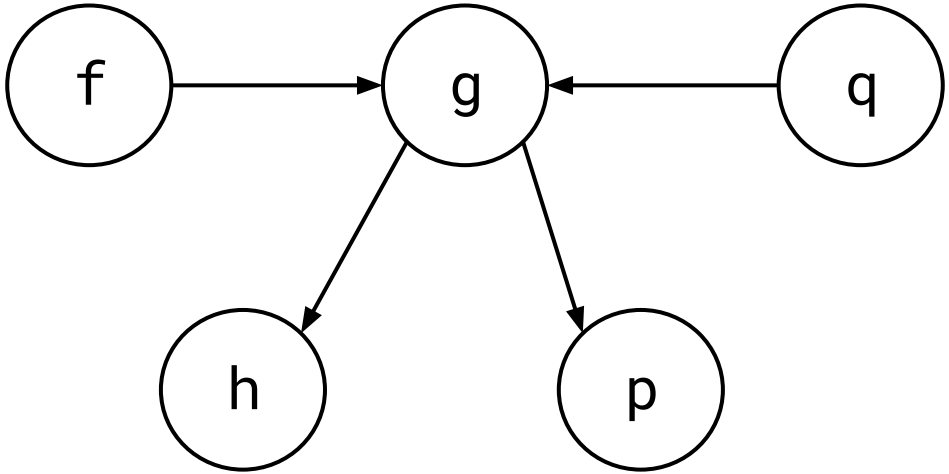
Let's arbitrarily pick f for now.



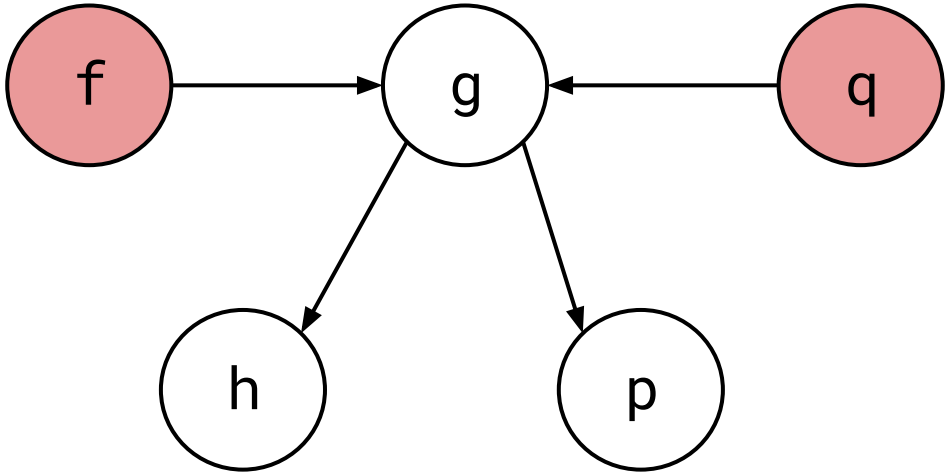
3) Delete edges pointing to loop breaker f



4) No more strongly connected components! YAY!



What does this mean? Recall our loop breakers q and f …



These are the identifiers we choose NEVER to inline.

Returning to our example…

```
recursiveGroup = f
    where
     f = (g 1)
     g = h . p
     h = \y -> if y < 15 then 5 else f
     q = g
     p = \x -> if x < 10 then 10 else q (x - 1)
```

Rearrange bindings in topological order

```
recursiveGroup = f
    where
     p = \x -> if x < 10 then 10 else q (x - 1)
     h = \y -> if y < 15 then 5 else f
     g = h . p
     f = (g 1) // loop breaker so never inlined
     q = g     // loop breaker so never inlined
```

Inline p …

```
recursiveGroup = f
    where
     p = \x -> if x < 10 then 10 else q (x - 1)
     h = \y -> if y < 15 then 5 else f
     g = h . (\x -> if x < 10 then 10 else q (x - 1))
     f = (g 1) // loop breaker so never inlined
     q = g     // loop breaker so never inlined
```

remove dead p…

```
recursiveGroup = f
    where

     h = \y -> if y < 15 then 5 else f
     g = h . (\x -> if x < 10 then 10 else q (x - 1))
     f = (g 1) // loop breaker so never inlined
     q = g     // loop breaker so never inlined
```

Inline h …

```
recursiveGroup = f
    where

      h = \y -> if y < 15 then 5 else f
      g = (\y -> if y < 15 then 5 else f) .
          (\x -> if x < 10 then 10 else q (x - 1))
      f = (g 1) // loop breaker so never inlined
      q = g     // loop breaker so never inlined
```

Remove dead h …

```
recursiveGroup = f
    where


    g = (\y -> if y < 15 then 5 else f) .
        (\x -> if x < 10 then 10 else q (x - 1))
    f = (g 1) // loop breaker so never inlined
    q = g     // loop breaker so never inlined
```

That’s as far as we’ll go.

```
recursiveGroup :: Int

recursiveGroup = f

    where

     g = (\y -> if y < 15 then 5 else f) .

         (\x -> if x < 10 then 10 else q (x - 1))

     f = (g 1)

     q = g
```

Probably would have been better to make g a loop breaker…
Heuristics would have helped us pick g.

To be continued!