A Gradual Typing Poem

Sam Tobin-Hochstadt & Robby Finder

The Problem

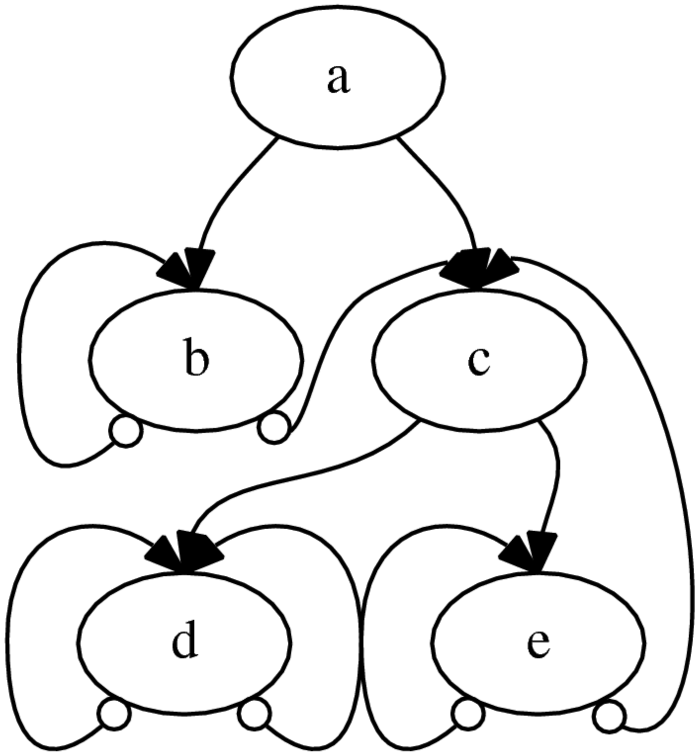
Write a function that accepts the specification of an infinite regular tree and turn it into a representation of the tree

The Problem

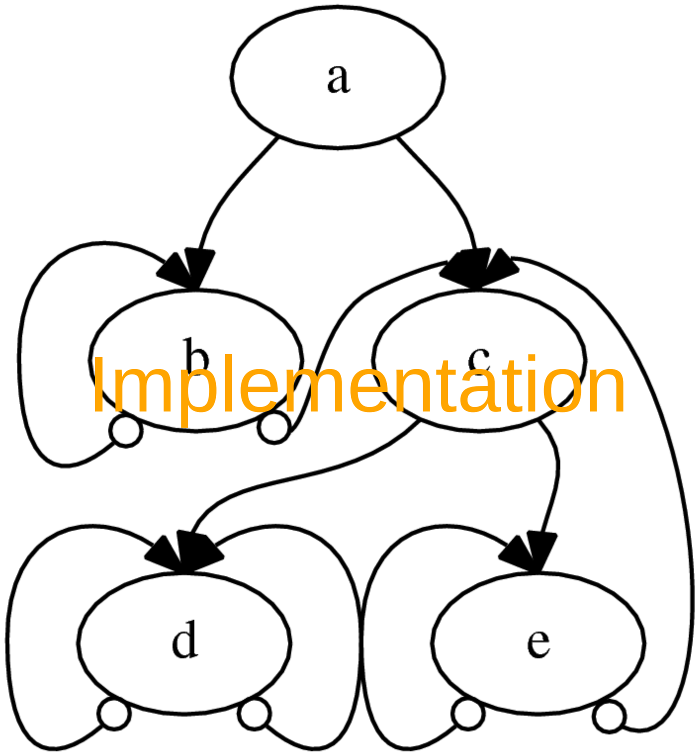
Write a function that accepts the specification of an infinite regular tree and turn it into a representation of the tree

Write a function that accepts a tree and finds its period

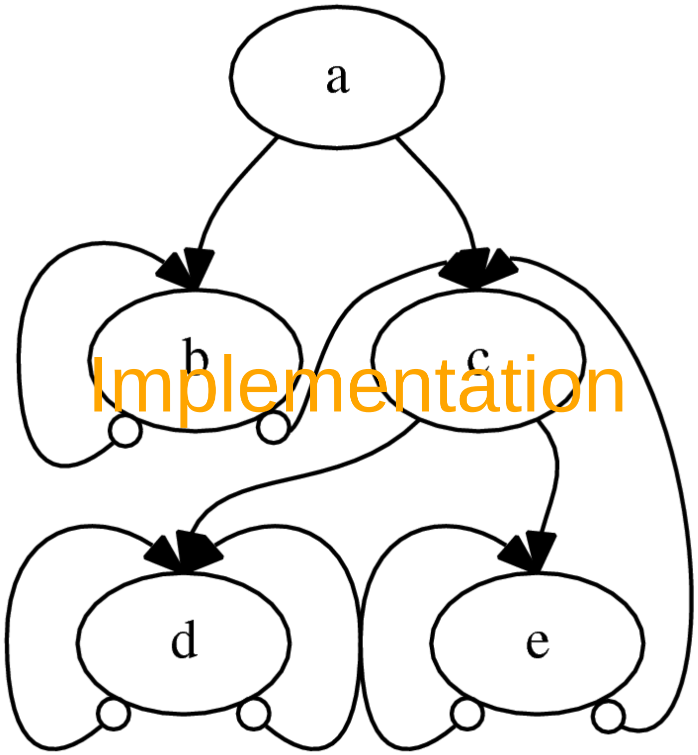
An Example



An Example

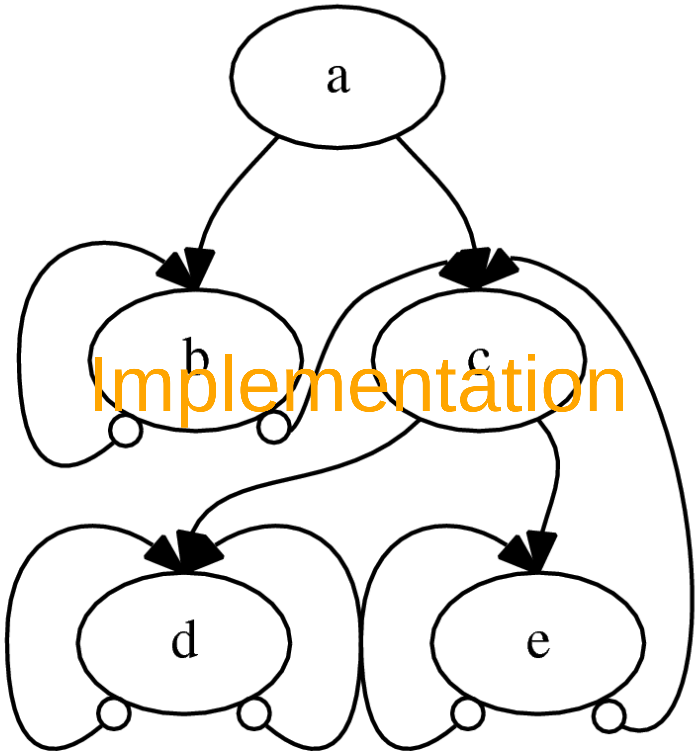


An Example




```
(a (b b c)
   (c (d d d)
      (e e c)))
```

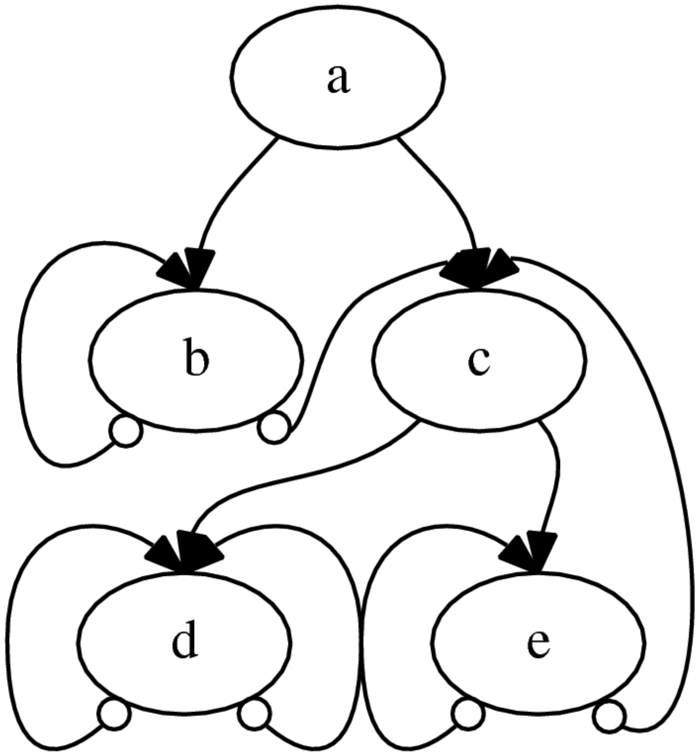
An Example




```
(a (b b c)
   (c (d d d)
      (e e c)))
```

Spec

How to Solve It




```
(a (b b c)
   (c (d d d)
      (e e c)))
```

The Problem with the Solution

The Standard Solution

- exposes mutability
- exposes placeholders
- pushes the burden onto the client (the period function)

The Problem with the Solution

```haskell
data STree = STree String STree STree | Link String
data ITree = ITree String ITree ITree

link :: STree -> ITree
link main = conv main
  where conv :: STree -> ITree
        conv (STree str tl tr) =
             ITree str (conv tl) (conv tr)
        conv (Link str) =
             find main str []

        find :: STree -> String -> [STree] -> ITree
        find (STree str2 tl tr) str pending
           | str2==str = conv (STree str2 tl tr)
           | otherwise = find tl str (tr:pending)
        find (Link str1) str2 (p:ps) = find p str2 ps

period :: ITree -> Maybe Int
period (ITree str tl tr) = bfs [(tl,1),(tr,1)] []
  where bfs :: [(ITree,Int)] -> [String] -> Maybe Int
        bfs [] visited = Nothing
        bfs ((ITree str2 tl tr,i) : rest) visited
         | str2==str = Just i
         | elem str2 visited = bfs rest visited
         | otherwise = bfs (rest ++ [(tl,i+1),(tr,i+1)])
                           (str2:visited)

left :: ITree -> ITree
left (ITree str l r) = l
right :: ITree -> ITree
right (ITree str l r) = r

at :: ITree
at = link (STree "a" (STree "b" (Link "b") (Link "c"))
                     (STree "c" (STree "d" (Link "d") (Link "d"))
                                (STree "e" (Link "e") (Link "c"))))

bt :: ITree
bt = left at

ct :: ITree
ct = right at

main :: IO ()
main = print [period at, period bt, period ct]
```

```sml
val exists=List.exists
val toString=Int.toString

datatype stree=STree of string * stree * stree | Link of string
datatype itree=ITree of string * (unit->itree) * (unit->itree)

(* link : stree -> itree *)
fun link main = let
  fun conv (STree (str,tl,tr)) =
      ITree (str,fn () => conv tl,fn () => conv tr)
    | conv (Link str) = find main str []
  and find (STree (str2,tl,tr)) str pending =
      if (str2=str)
      then conv (STree (str2,tl,tr))
      else find tl str (tr::pending)
   | find (Link str1) str2 (p::ps) = find p str2 ps
in conv main end


(* period : ITree -> int option *)
fun period (ITree (str,tl,tr)) = let
  fun elem n l = exists (fn x => (n = x)) l
  fun bfs [] visited = NONE
    | bfs ((ITree (str2,tl,tr),i) :: rest) visited =
      if (str2=str) then SOME i else
      if (elem str2 visited) then bfs rest visited
      else bfs (rest @ [(tl(),i+1),(tr(),i+1)]) (str2::visited)
in bfs [(tl(),1),(tr(),1)] [] end

val at = link (STree ("a",STree("b",Link "b", Link "c"),
                          STree("c",STree("d",Link "d", Link "d"),
                                    STree("e",Link "e", Link "c"))))

fun left (ITree (st,tl,tr)) = tl()
fun right (ITree (st,tl,tr)) = tr()

val bt = left at
val ct = right at

val answers = [period at, period bt, period ct]

val change_up = let
  val r = ref 0
  fun f () = (r := !r+1; ITree (toString (!r),f,f))
in ITree("a",f,f) end
```

```sml
val exists=List.exists

datatype stree = STree of string * stree * stree | Link of stri

datatype itree = ITree of string *
                 itree option ref *
                 itree option ref

(* link : stree -> itree *)
fun link main = let
  val trees = ref []     val tolink = ref []
  fun conv (STree (str,tl,tr)) = let
      val t = ITree (str,conv tl,conv tr)
      in trees := t :: !trees; ref (SOME t) end
    | conv (Link str) = let
      val r = ref NONE
      in tolink := (r,str) :: !tolink; r end
  val ans = conv main
in app (fn (ITree (str,tl,tr)) =>
       app (fn (r,str2) =>
             if str = str2
             then r:=SOME (ITree (str,tl,tr))
             else ())
           (!tolink))
       (!trees);
  case !ans of SOME x => x
end


(* period : ITree -> int option *)
fun period (ITree (str,ref (SOME tl),ref (SOME tr))) = let
  fun elem (n:string) l = exists (fn x => (n = x)) l
  fun help [] visited = NONE
    | help ((ITree (str2,ref (SOME tl),ref (SOME tr)),i) :: res
           visited =
      if (str2=str) then SOME i else
      if (elem str2 visited) then help rest visited
      else help (rest @ [(tl,i+1),(tr,i+1)]) (str2::visited)
in
  help [(tl,0),(tr,0)] []
end

val at = link (STree ("a",STree("b",Link "b", Link "c"),
                          STree("c",STree("d",Link "d", Link "d
                                    STree("e",Link "e", Link "c

fun left (ITree (st,ref (SOME tl),ref (SOME tr))) = tl
fun right (ITree (st,ref (SOME tl),ref (SOME tr))) = tr

val bt = left at
val ct = right at

val answers = [period at, period bt, period ct]
```

- **Problem Statement**
- **The Typed Scheme Advantage**
- **An Intro to Typed Scheme**
- **The First Solution**
- **The Second Solution**
- **The Moral**

Where We’re Going

Typed Scheme allows a simple implementation of the problem where

- the complexity of the implementation is hidden
- the client has all the advantages of the original code

Where We're Going

Typed Scheme allows a simple implementation of the problem where

- the complexity of the implementation is hidden
- the client has all the advantages of the original code

All because of gradual typing!

- Problem Statement
- The Typed Scheme Advantage
- **An Intro to Typed Scheme**
- The First Solution
- The Second Solution
- The Moral

Typed Scheme

```
#lang typed-scheme

(: x Number)
(define x 1)
```

Typed Structs

```
#lang typed-scheme

(define-struct: ImpTree
  ([name  : Symbol]
   [left  : (U ImpTree Symbol)]
   [right : (U ImpTree Symbol)]))
```

Occurrence Typing

```
#lang typed-scheme

(if (ImpTree? t)
    (display (ImpTree-name t))
    (display "no name"))
```

Modules

```
#lang typed-scheme

(: t ImpTree)
(define t (make-ImpTree 'a 'x 'y))
(provide t)
```

Typed/Untyped Integration

```
#lang scheme

(provide t)
(define t (make-ImpTree ))
```

contract boundary

```
#lang typed-scheme

(require/typed "x.ss" [t ImpTree])
(ImpTree-left t)
```

- Problem Statement
- The Typed Scheme Advantage
- An Intro to Typed Scheme
- The First Solution
- The Second Solution
- The Moral

Specification

```
(define-type-alias SpecTree
  (Rec ST (U Symbol (List Symbol ST ST))))
```

Specification

```
(define-type-alias SpecTree
  (Rec ST (U Symbol (List Symbol ST ST))))
(define-struct: ImpTree
  ([name  : Symbol]
   [left  : (U ImpTree Symbol)]
   [right : (U ImpTree Symbol)])
  #:mutable)
```

Client Code

```
(: period (ImpTree -> (U Number #f)))
(define (period it)              )
```

Client Code

```
(: period (ImpTree -> (U Number #f)))
(define (period it)
  (: bfs        )
  (define (bfs s v)        )
  (let ([l (ImpTree-left it)]
        [r (ImpTree-right it)])
    (if (and (ImpTree? l) (ImpTree? r))
        (bfs (list (cons l 1) (cons r 1))
             '())
        (error 'fail))))
```

Client Code

```
(: period (ImpTree -> (U Number #f)))
(define (period it)
  (: bfs )
  (define (bfs s v) )
  (let ([l (ImpTree-left it)]
        [r (ImpTree-right it)])
    (if (and (ImpTree? l) (ImpTree? r))
        (bfs (list (cons l 1) (cons r 1))
             '())
        (error 'fail))))
```

Client Code

```
(: bfs ((Listof (Pair ImpTree Number))
        (Listof Symbol)
        -> (U Number #f)))
(define (bfs stack visited)
  (match stack
    ['() #f]
    [(cons (cons (struct ImpTree (str2 tl tr)) i)
           rest)
     (cond
       [(eq? str2 (ImpTree-name it)) i]
       [(memq str2 visited) (bfs rest visited)]
       [(and (ImpTree? tl) (ImpTree? tr))
        (bfs (append rest (list (cons tl (add1 i))
                                (cons tr (add1 i))))
             (cons str2 visited))]
       [else (error 'fail)])]))
```

Client Code

Exactly the problem we thought we'd have

- Problem Statement
- The Typed Scheme Advantage
- An Intro to Typed Scheme
- The First Solution
- The Second Solution
- The Moral

Better Specification

```
(define-type-alias SpecTree
  (Rec ST (U Symbol (List Symbol ST ST))))
(define-struct: ImpTree
  ([name  : Symbol]
   [left  : ImpTree]
   [right : ImpTree]))
```

Gradual Typing to the Rescue

```
(require/typed "itree.ss"
  [struct ImpTree ([name  : Symbol]
                   [left  : ImpTree]
                   [right : ImpTree])]
  [link (SpecTree -> ImpTree)])
```

Client Code

```
(: period (ImpTree -> (U Number #f)))
(define (period it)          )
```

Client Code

```
(: period (ImpTree -> (U Number #f)))
(define (period it)
  (let ([l (ImpTree-left it)]
        [r (ImpTree-right it)])
    (bfs (list (cons l 1) (cons r 1))
         '())))
```

Client Code

```
(: bfs ((Listof (Pair ImpTree Number))
        (Listof Symbol)
        -> (U Number #f)))
(define (bfs stack visited)
  (match stack
    ['() #f]
    [(cons (cons (struct ImpTree (str2 tl tr)) i)
           rest)
     (cond
       [(eq? str2 (ImpTree-name it)) i]
       [(memq str2 visited) (bfs rest visited)]
       [else
        (bfs (append rest
                     (list (cons tl (add1 i))
                           (cons tr (add1 i))))
             (cons str2 visited))])]))
```

What Happened?

Typed Scheme automatically synthesized contracts

Mutation is hidden

- Problem Statement
- The Typed Scheme Advantage
- An Intro to Typed Scheme
- The First Solution
- The Second Solution
- The Moral

Moral

Gradual Typing adds expressiveness to ***typed*** languages

Thank You