Programación basada en pruebas unitarias

Miguel Lechón

28 de diciembre de 2014

Contexto – SigBOVIK

SigBOVIK 2014

2013 2012 2011 2010 2009 2008 2007

ACH HOMEPAGE | LISTSERV | EASYCHAIR SUBMISSIONS

You Won't Believe This Call For Papers

This 8th annual ACH Special Interest Group on Harry Quicksand Bovik will blow your mind.

-1. Awards

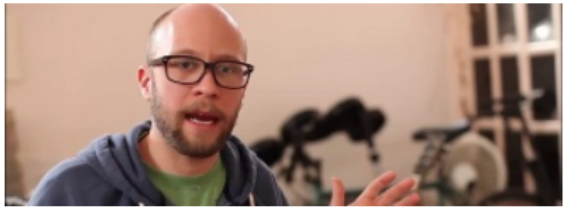
Most Frighteningly Like Real Research — Miguel Lechón
Unit-Test-Based Programming

Most Deserving of Being Real Research — Tom Murphy VII
New Results in k/n Power Hours

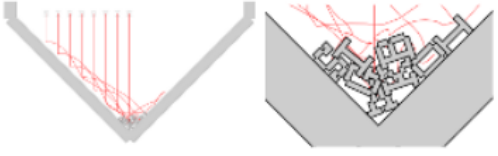
Most Likely to Get Us Featured on Fox News — Stefan Muller
Heterotopy Type Theory

Most Likely to Save the Universe — Jenn Landefeld
Please Don't Let Open House Destroy the Universe

Popular SIGBOVIK Research



This Man Teaches Computers To Play Mario. What Happens Next Will Amaze You



DIY: Constructing A Trapdoor Function With Physics In N Simple Timesteps

Contexto – Yo

- Informático

Contexto – Yo

- Informático
- Programador sin código en producción

Contexto – Yo

- Informático
- Programador sin código en producción
- Administrador de sistemas mediocre

Contexto – Yo

- Informático
- Programador sin código en producción
- Administrador de sistemas mediocre
- Ex-estudiante de doctorado no doctorado

Contexto – Yo

- Informático
- Programador sin código en producción
- Administrador de sistemas mediocre
- Ex-estudiante de doctorado no doctorado
- Actualmente, ama de casa

Motivación

¿En qué consiste programar?

```
def factorial(n):
    """
    Returns the factorial of n
    (int -> int)

    >>> factorial(2)
    2
    >>> factorial(3)
    6
    """
    if n:
        return n*factorial(n-1)
    else:
        return 1
```

Motivación

¿En qué consiste programar?

```
def factorial(n):
    """
    Returns the factorial of n
    (int -> int)

    >>> factorial(2)
    2
    >>> factorial(3)
    6
    """
    if n:
        return n*factorial(n-1)
    else:
        return 1
```

- Explicar qué se espera del código

Motivación

¿En qué consiste programar?

```
def factorial(n):
    """
    Returns the factorial of n
    (int -> int)

    >>> factorial(2)
    2
    >>> factorial(3)
    6
    """
    if n:
        return n*factorial(n-1)
    else:
        return 1
```

- Explicar qué se espera del código
- Explicarlo otra vez, por si las moscas

Motivación

¿En qué consiste programar?

```
def factorial(n):
    """
    Returns the factorial of n
    (int -> int)

    >>> factorial(2)
    2
    >>> factorial(3)
    6
    """
    if n:
        return n*factorial(n-1)
    else:
        return 1
```

- Explicar qué se espera del código
- Explicarlo otra vez, por si las moscas
- Explicarlo de nuevo, esta vez para el ordenador

Motivación

¿En qué consiste programar?

```
def factorial(n):
    """
    Returns the factorial of n
    (int -> int)

    >>> factorial(2)
    2
    >>> factorial(3)
    6
    """
    if n:
        return n*factorial(n-1)
    else:
        return 1
```

- Explicar qué se espera del código
- Explicarlo otra vez, por si las moscas
- Explicarlo de nuevo, esta vez para el ordenador

Nadie aprende a programar para trabajar más de la cuenta.
Tiene que haber una manera mejor…

Programación basada en pruebas unitarias

```
@UTBP
def factorial(n):
    """
    factorial(0) == 1
    factorial(1) == 1
    factorial(2) == 2
    factorial(3) == 6
    """
```

Programación basada en pruebas unitarias

```
@UTBP
def factorial(n):
    """
    factorial(0) == 1
    factorial(1) == 1
    factorial(2) == 2
    factorial(3) == 6
    """
```

Sí, funciona.

Pero... ¿cómo?

Premisas:

Pero... ¿cómo?

Premisas:

- Toda función se puede escribir de forma recursiva

Pero... ¿cómo?

Premisas:

- Toda función se puede escribir de forma recursiva
- Toda función recursiva puede representarse mediante un árbol

Pero... ¿cómo?

Premisas:

- Toda función se puede escribir de forma recursiva
- Toda función recursiva puede representarse mediante un árbol
- Los árboles se pueden enumerar en orden creciente de complejidad

PERO... ¿CÓMO?

Premisas:

- Toda función se puede escribir de forma recursiva
- Toda función recursiva puede representarse mediante un árbol
- Los árboles se pueden enumerar en orden creciente de complejidad

Conclusión:
$\rightarrow$ **Las funciones se pueden enumerar en orden creciente de complejidad**

Pero... ¿cómo?

Premisas:

- Toda función se puede escribir de forma recursiva
- Toda función recursiva puede representarse mediante un árbol
- Los árboles se pueden enumerar en orden creciente de complejidad

Conclusión:
$\rightarrow$ **Las funciones se pueden enumerar en orden creciente de complejidad**
(y podemos comprobar, una a una, si satisfacen un conjunto arbitrario de pruebas unitarias)

RECURSIVIDAD PARA TODO

Advertencia: Este es un ejercicio académico

RECURSIVIDAD PARA TODO

Advertencia: Este es un ejercicio académico extremadamente complejo

RECURSIVIDAD PARA TODO

Advertencia: Este es un ejercicio académico extremadamente complejo. Gritad si os perdéis.

RECURSIVIDAD PARA TODO

Advertencia: Este es un ejercicio académico extremadamente complejo. Gritad si os perdéis.

```
def length(l):
    i = 0
    for e in l:
        i += 1
    return i
```

RECURSIVIDAD PARA TODO

Advertencia: Este es un ejercicio académico extremadamente complejo. Gritad si os perdéis.

```
def length(l):
    i = 0
    for e in l:
        i += 1
    return i
```

```
def length(l):
    if l:
        return 1+length(l[1:])
    else:
        return 0
```

RECURSIVIDAD PARA TODO

Advertencia: Este es un ejercicio académico extremadamente complejo. Gritad si os perdéis.

```
def length(l):
    i = 0
    for e in l:
        i += 1
    return i
```

```
def length(l):
    if l:
        return 1+length(l[1:])
    else:
        return 0
```

La versión recursiva es claramente superior.

Recursividad para todo

Advertencia: Este es un ejercicio académico extremadamente complejo. Gritad si os perdéis.

```
def length(l):
    i = 0
    for e in l:
        i += 1
    return i
```

```
def length(l):
    if l:
        return 1+length(l[1:])
    else:
        return 0
```

La versión recursiva es claramente superior.

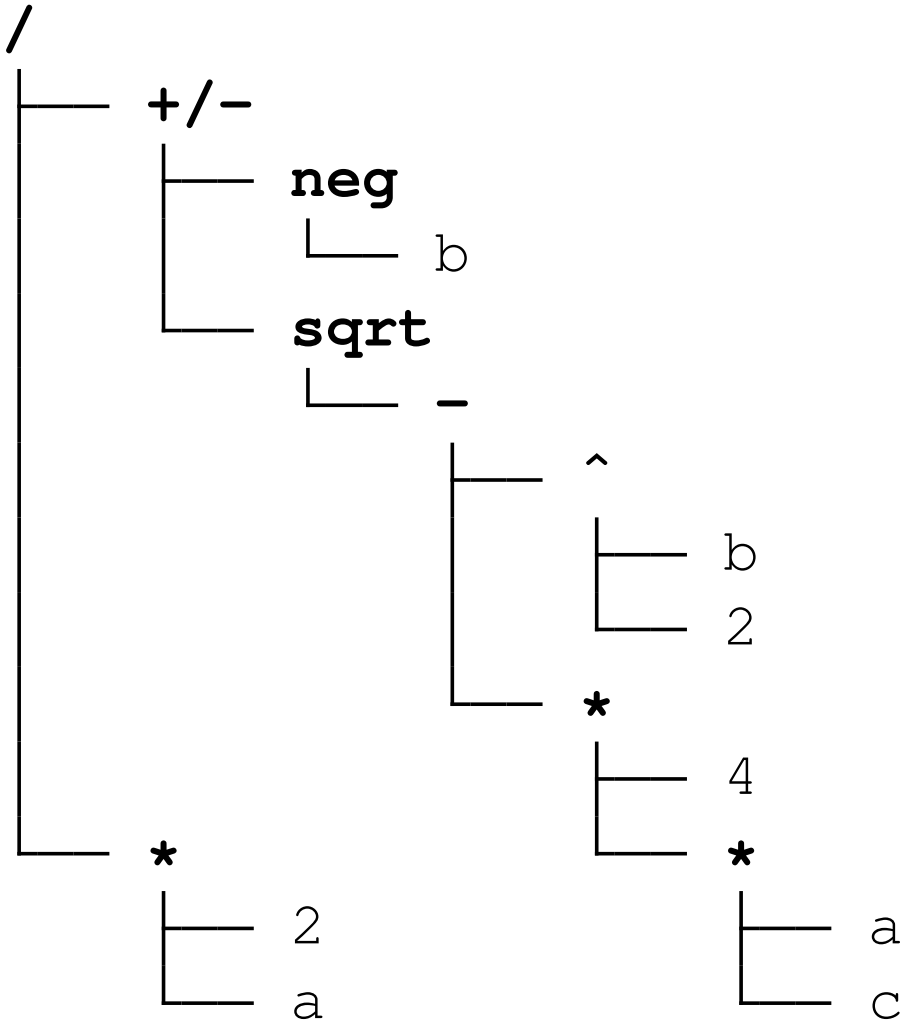
```
length = lambda l: sum(1 for e in l)
```

Expresiones como árboles

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

EXPRESIONES COMO ÁRBOLES

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



FUNCIONES RECURSIVAS COMO ÁRBOLES

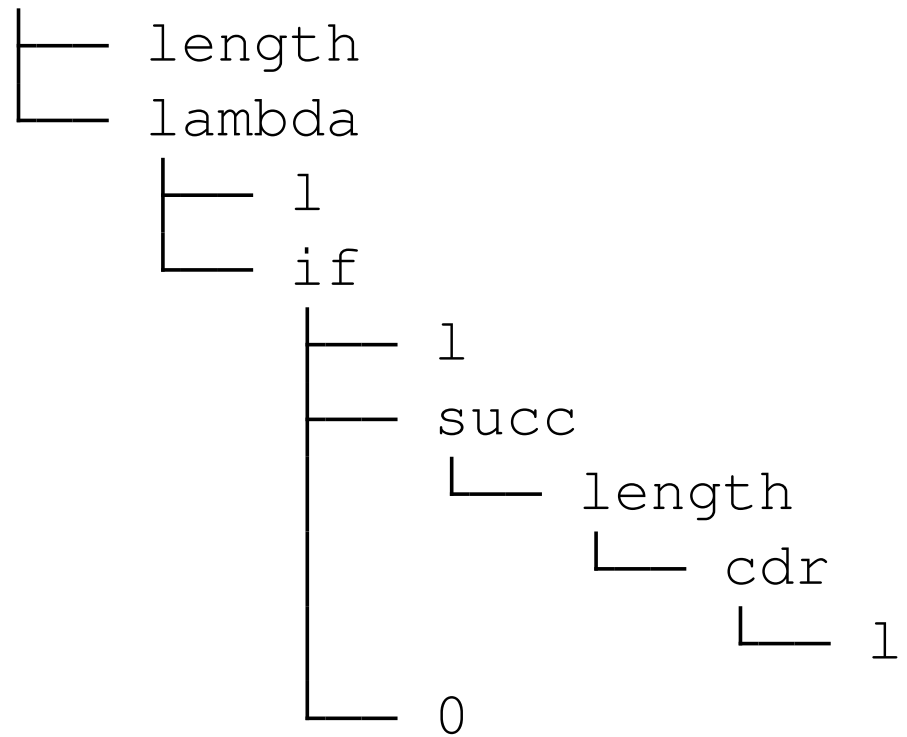
```
def length(l):
    if l: return 1+length(l[1:])
    else: return 0
```

python

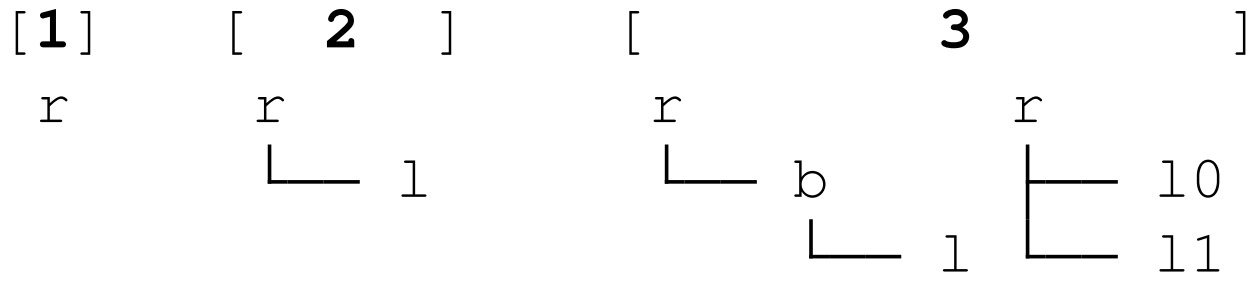
```
def
  length

(l):
    if
      l:
        return 1+
            length(
                l[1:]
            )
    else: return 0
```

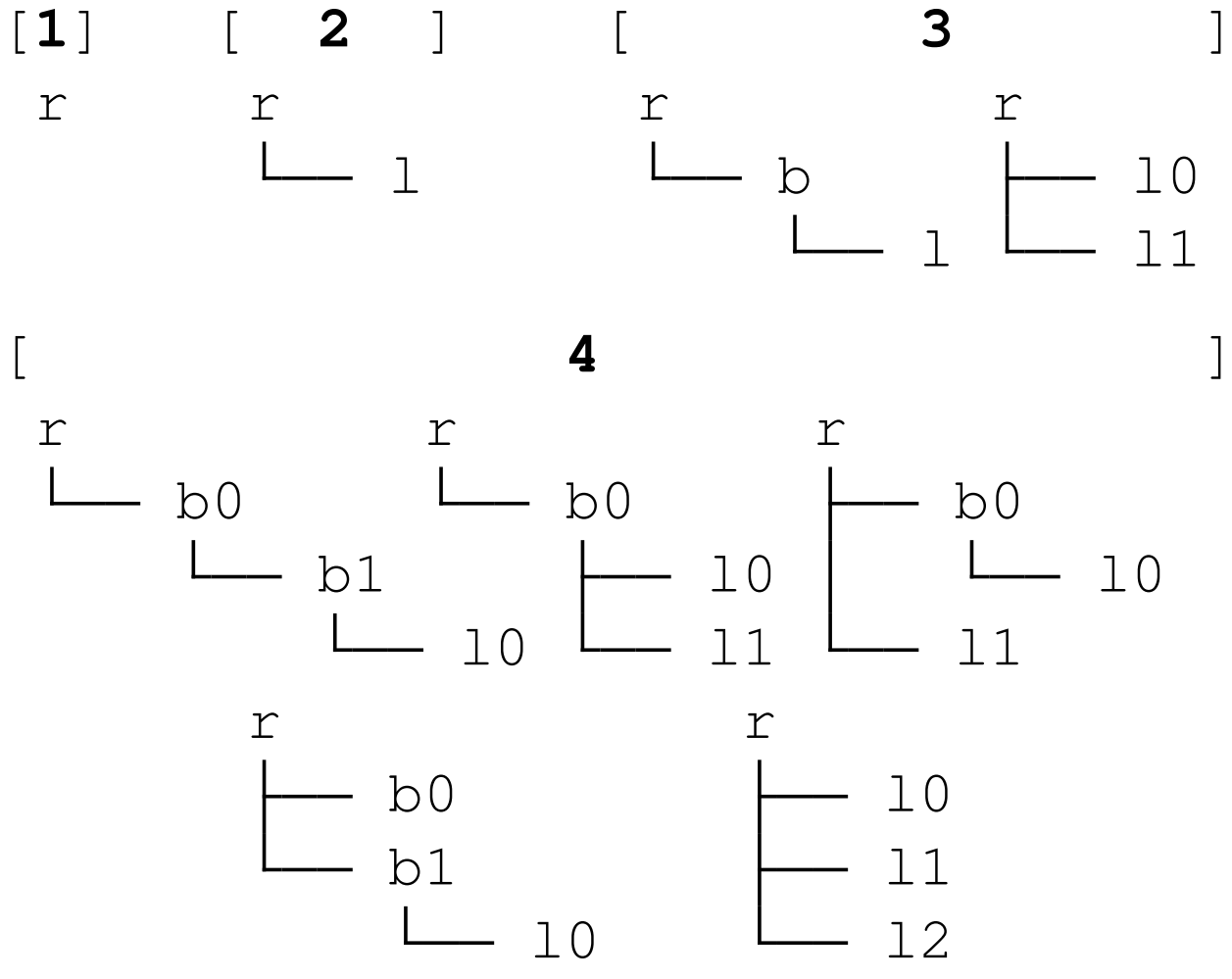
pseudo-scheme



Enumeración de árboles



Enumeración de árboles



INGREDIENTES DEL PARADIGMA UTBP

- Estructuras arborescentes

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*
- Pobladas por operaciones:

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*
- Pobladas por operaciones:
  - Funciones aritméticas (`succ`, `pred`)

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*
- Pobladas por operaciones:
  - Funciones aritméticas (`succ`, `pred`)
  - Manipulación de listas (`car`, `cdr`, `cons`, `list?`)

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*
- Pobladas por operaciones:
  - Funciones aritméticas (`succ`, `pred`)
  - Manipulación de listas (`car`, `cdr`, `cons`, `list?`)
  - Formas especiales (`if`)

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*
- Pobladas por operaciones:
  - Funciones aritméticas (`succ`, `pred`)
  - Manipulación de listas (`car`, `cdr`, `cons`, `list?`)
  - Formas especiales (`if`)
- Sobre dos tipos:

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*
- Pobladas por operaciones:
  - Funciones aritméticas (`succ`, `pred`)
  - Manipulación de listas (`car`, `cdr`, `cons`, `list?`)
  - Formas especiales (`if`)
- Sobre dos tipos:
  - Números naturales no negativos

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*
- Pobladas por operaciones:
  - Funciones aritméticas (`succ`, `pred`)
  - Manipulación de listas (`car`, `cdr`, `cons`, `list?`)
  - Formas especiales (`if`)
- Sobre dos tipos:
  - Números naturales no negativos
  - Listas, posiblemente anidadas (realmente tuplas, porque no pensamos mutarlas)

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*
- Pobladas por operaciones:
  - Funciones aritméticas (`succ`, `pred`)
  - Manipulación de listas (`car`, `cdr`, `cons`, `list?`)
  - Formas especiales (`if`)
- Sobre dos tipos:
  - Números naturales no negativos
  - Listas, posiblemente anidadas (realmente tuplas, porque no pensamos mutarlas)
- Constantes (operaciones sobre cero elementos):

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*
- Pobladas por operaciones:
  - Funciones aritméticas (`succ`, `pred`)
  - Manipulación de listas (`car`, `cdr`, `cons`, `list?`)
  - Formas especiales (`if`)
- Sobre dos tipos:
  - Números naturales no negativos
  - Listas, posiblemente anidadas (realmente tuplas, porque no pensamos mutarlas)
- Constantes (operaciones sobre cero elementos):
  - `0`

Ingredientes del paradigma UTBP

- Estructuras arborescentes *(déjà vu)*
- Pobladas por operaciones:
  - Funciones aritméticas (`succ`, `pred`)
  - Manipulación de listas (`car`, `cdr`, `cons`, `list?`)
  - Formas especiales (`if`)
- Sobre dos tipos:
  - Números naturales no negativos
  - Listas, posiblemente anidadas (realmente tuplas, porque no pensamos mutarlas)
- Constantes (operaciones sobre cero elementos):
  - `0`
  - `()`

EJEMPLO DE USO – LONGITUD DE UNA LISTA

Ejemplo de uso – Longitud de una lista

```
@UTBP
def length(l):
"""
length(()) == 0
"""
```

EJEMPLO DE USO – LONGITUD DE UNA LISTA

```
@UTBP
def length(l):
"""
length(()) == 0
"""
```

```
define
├── length
└── lambda
    ├── l
    └── 0
```

Ejemplo de uso – Longitud de una lista

```
@UTBP
def length(l):
"""
length(()) == 0
"""
```

```
define
├── length
└── lambda
    ├── l
    └── 0
```

```
@UTBP
def length(l):
"""
length(()) == 0
length((2,)) == 1
"""
```

Ejemplo de uso – Longitud de una lista

```
@UTBP
def length(l):
"""
length(()) == 0
"""
```

```
define
├── length
└── lambda
    ├── l
    └── 0
```

```
@UTBP
def length(l):
"""
length(()) == 0
length((2,)) == 1
"""
```

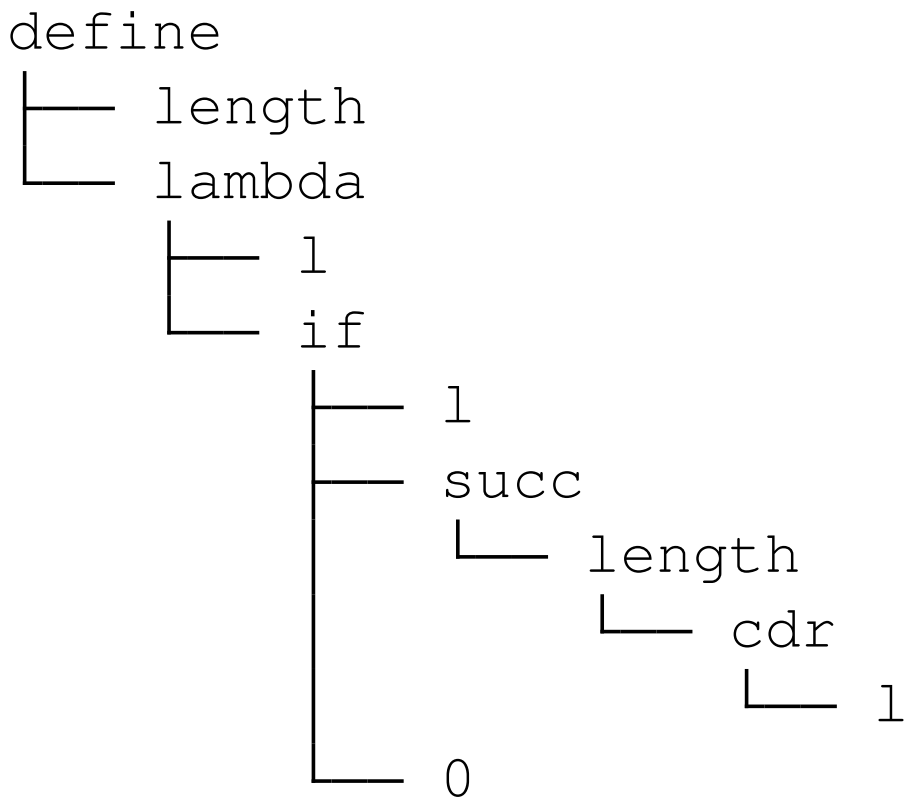
```
define
├── length
└── lambda
    ├── l
    └── if
        ├── l
        ├── list?
        │   └── l
        └── 0
```

Ejemplo de uso – Longitud de una lista

```
@UTBP
def length(l):
"""
length(()) == 0
length((2,)) == 1
length((2, 3)) == 2
"""
```

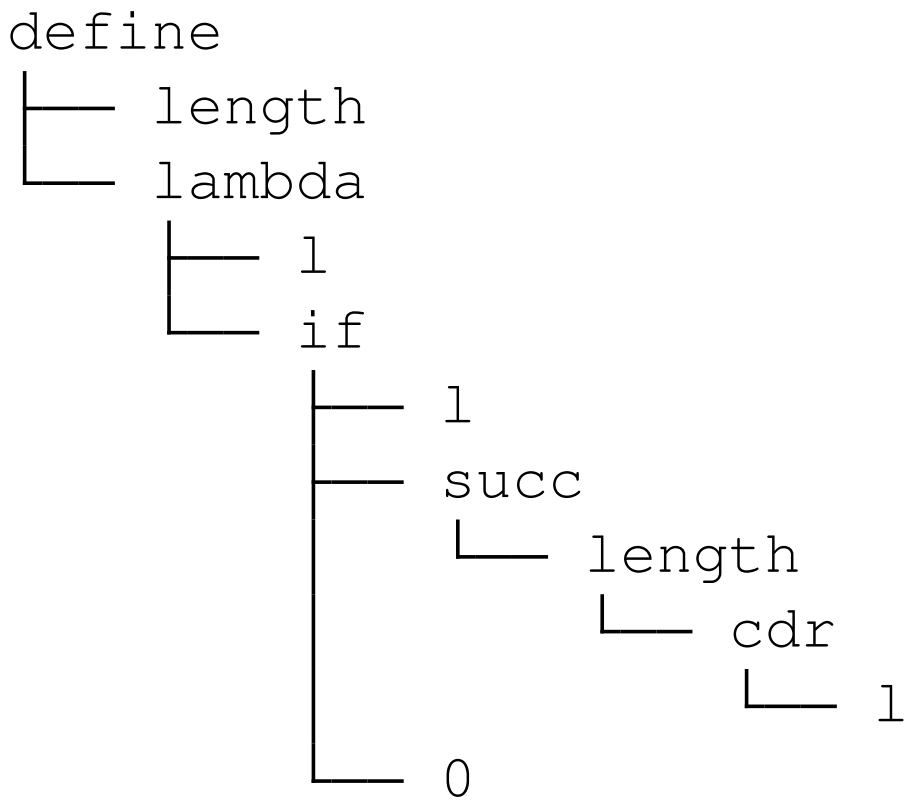
Ejemplo de uso – Longitud de una lista

```
@UTBP
def length(l):
"""
length(()) == 0
length((2,)) == 1
length((2, 3)) == 2
"""
```



Ejemplo de uso – Longitud de una lista

```
@UTBP
def length(l):
"""
length(()) == 0
length((2,)) == 1
length((2, 3)) == 2
"""
```



Glorioso.

Características

- Corrección garantizada

Características

- Corrección garantizada

```
@UTBP
def imposible(l):
"""
imposible(0) == 0
imposible(0) == 1
"""
```

Características

- Corrección garantizada

```
@UTBP
def imposible(l):
"""
imposible(0) == 0
imposible(0) == 1
"""
```

- Prevención de días del juicio final del estilo *Terminator* (amurallado dentro de un intérprete de Scheme derivado de `http://norvig.com/lispy.html`)

Características

- Corrección garantizada

```
@UTBP
def imposible(l):
"""
imposible(0) == 0
imposible(0) == 1
"""
```

- Prevención de días del juicio final del estilo *Terminator* (amurallado dentro de un intérprete de Scheme derivado de `http://norvig.com/lispy.html`)
- Interoperable con Python estándar

Características

- Corrección garantizada

```
@UTBP
def imposible(l):
"""
imposible(0) == 0
imposible(0) == 1
"""
```

- Prevención de días del juicio final del estilo *Terminator* (amurallado dentro de un intérprete de Scheme derivado de `http://norvig.com/lispy.html`)
- Interoperable con Python estándar
- Complejidad de Kolmogorov mínima

Características

- Corrección garantizada

```
@UTBP
def imposible(l):
"""
imposible(0) == 0
imposible(0) == 1
"""
```

- Prevención de días del juicio final del estilo *Terminator* (amurallado dentro de un intérprete de Scheme derivado de `http://norvig.com/lispy.html`)
- Interoperable con Python estándar
- Complejidad de Kolmogorov mínima
- Navajitud de Occam máxima

Conclusión

La programación imperativa es como muy del siglo XX.

Conclusión

La programación imperativa es como muy del siglo XX.
El mundo necesita herramientas más simples y estúpidas para programadores descuidados.

Conclusión

La programación imperativa es como muy del siglo XX.
El mundo necesita herramientas más simples y estúpidas para programadores descuidados.

¿Preguntas?