```
        (e / f -> \v.e0*)  => z
EAGER  ----------------------------        e0* = letrec f = \x. e0 in e0
        letrec f = \v. e0 in e => z
```

```
        e (rec e) => z
NORMAL  --------------
        rec e => z
```

```
letrec f = \v. e0 in e   =def   let f = rec(\f.\v.e0) in e   =def  (\f.e)(rec(\f.\v.e0))
```

De donde se deduce la regla:

```
        (e / f -> rec(\f.\v.e0))  => z
NORMAL  ------------------------------
        letrec f = \v. e0 in e => z
```

En las páginas siguientes se muestran las dos derivaciones del factorial

```
letrec fact = \x. if x=0 then 1 else x * f (x-1) in f 2
(f 2 / f -> \x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1))
(\x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1)) 2
      |_\x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1) => simismo
      |_2 => 2
      |letrec fact = \x. e0 in if 2=0 then 1 else 2 * f (2-1)   (ahora se ahorran pasos en la aplicación de letrec)
      |if 2=0 then 1 else 2 * (\x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1)) (2-1)
      |2 * (\x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1)) (2-1)
      |     |_2 => 2
      |     |(\x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1)) (2-1)
      |     |     |_(\x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1)) => simismo
      |     |     |_2-1 => 1
      |     |     |letrec fact = \x. e0 in if 1=0 then 1 else 1 * f (1-1)
      |     |     |if 1=0 then 1 else 1 * (\x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1)) (1-1)
      |     |     |1 * (\x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1)) (1-1)
      |     |     |     |_1 => 1
      |     |     |     |_(\x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1)) (1-1)
      |     |     |     |     |_(\x. letrec fact = \x. e0 in if x=0 then 1 else x * f (x-1)) => simismo
      |     |     |     |     |_1-1 => 0
      |     |     |     |     |letrec fact = \x. e0 in if 0=0 then 1 else 0 * f (0-1)
      |     |     |     |     |if 0=0 then 1 else 0 * f (0-1)
      |     |     |     |     |_=> 1
      |     |     |     |_=> 1
      |     |     |_=>1
      |     |_=>1
      |_=>2
=> 2
```

```
letrec fact = \x. if x=0 then 1 else x * f (x-1) in f 2
(f 2 / f -> rec(\f.\x.e0))
rec(\f.\x.e0) 2
      |rec(\f.\x.e0)
      |(\f.\x.e0) (rec(\f.\x.e0))
      |_=> (\x.if x=0 then 1 else x * rec(\f.\x.e0) (x-1))
      |if 2=0 then 1 else 2 * rec(\f.\x.e0) (2-1)
      |2 * rec(\f.\x.e0) (2-1)
      |     |_2 => 2
      |     |rec(\f.\x.e0) (2-1)
      |     | …
      |     |     |if 2-1=0 then 1 else (2-1) * rec(\f.\x.e0) ((2-1)-1)
      |     |     |(2-1) * rec(\f.\x.e0) ((2-1)-1)
      |     |     |_2-1 => 1
      |     |     |rec(\f.\x.e0) ((2-1)-1)
      |     |     | …
      |     |     |     |if (2-1)-1=0 then 1 else (2-1)-1 * rec(\f.\x.e0) ((2-1)-1-1)
      |     |     |     |_=>1
      |     |     |_=> 1
      |     |_=> 1
      |_=> 2
=> 2
```

```
newvar x := 2 in while val x /= 0 do x := val x - 2
let x = ref 2 in letrec mod = \v. if val x /= 0 then x := val x - 2; mod v else <> in mod <>
s, (\x. letrec mod = \v. if val x /= 0 then x := val x - 2; mod v else <> in mod <>) (ref 2)
s, (\x. letrec mod = \v. if val x /= 0 then x := val x - 2; mod v else <> in mod <>) => simismo
s, ref 2 => r,[s|r:2]
                        Definimos e0 = if val r /= 0 then r := val r - 2; mod v else <>
[s|r:2], letrec mod = \v. e0 in mod <>
[s|r:2], (mod <>) (\v. Letrec mod = e0 in if val r /= 0 then r := val r - 2; mod v else <>)
[s|r:2], (\v. Letrec mod = e0 in if val r /= 0 then r := val r - 2; mod v else <>) <>
|_[s|r:2], (\v. Letrec mod = e0 in if val r /= 0 then r := val r - 2; mod v else <>) => simismo
|_[s|r:2], <> => simismo
|[s|r:2], letrec mod = e0 in if val r /= 0 then r := val r - 2; mod <> else <> in mod <>
...
|[s|r:2], if val r /= 0 then r := val r - 2; (\v.e0*) <> else <>
|     [s|r:2], val r /= 0
|           [s|r:2], val r => 2,[s|r:2]
|           [s|r:2], 0 => 0,[s|r:2]
|     => true,[s|r:2]
|     [s|r:2],  r := val r - 2; (\v.e0*) <>
|           [s|r:2],  r := val r - 2 => <>, [s|r:0]
|           [s|r:0],  (\v.e0*) <>
|                 …
|                 [s|r:0],  if val r /= 0 then r := val r - 2; (\v.e0*) <> else <>
|                       [s|r:0],  val r /= 0
|                       ...
|                       => false, [s|r:0]
|                       => <>, [s|r:0]
|                 => <>, [s|r:0]
|           => <>, [s|r:0]
```

```
|     => <>, [s|r:0]
=> <>, [s|r:0]
```

DEDUCCIÓN DE LA REGLA PARA WHILE

Sea e0 = if b then e; w v else <>

Evaluar while b do e, que se traduce como letrec w = \v. e0 in w <>, es necesario evaluar (\v.e0*) <>. Pero para esto debemos evaluar

```
      letrec w = \v. e0 in if b then e; w <> else <>
```

Que a su vez requiere evaluar

```
      if b then e; (\v.e0*) <> else <>
```

Si reescribimos esta expresión, utilizando “las equivalencias” while b do e = (\v.e0*) <> y skip = <>, tenemos

```
      if b then e; while b do e else skip
```

Esto nos permite establecer las reglas:

```
s,b => false,s’
--------------------------------------------
while b do e => skip,s’

s,b => true,s’     s’, e;while b do e => z,s’’
--------------------------------------------
while b do e => z,s’’
```