Continuations: Exceptions, backtracking, Micro-Icon

David Christiansen¹

Monday, 29 October, 2012

¹based on slides by Peter Sestoft
Overview

CONTINUATIONS AND CONTINUATION-PASSING STYLE
Stack frames and continuations
Continuation-passing style
Tail recursion and iteration
CPS in Java

IMPLEMENTING EXCEPTIONS
Throwing exceptions
Handling exceptions

MICRO-ICON
Micro-Icon introduction
Micro-Icon interpreter
Overview

CONTINUATIONS AND CONTINUATION-PASSING STYLE
Stack frames and continuations
Continuation-passing style
Tail recursion and iteration
CPS in Java

IMPLEMENTING EXCEPTIONS
Throwing exceptions
Handling exceptions

MICRO-ICON
Micro-Icon introduction
Micro-Icon interpreter
let rec facr n =
    if n = 0
    then 1
    else n * facr (n - 1)

    facr 3
let rec facr n =
    if n = 0
    then 1
    else n * facr (n - 1)

    facr 3
⇒ 3 * facr (3 - 1)
let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

  facr 3
⇒ 3 * facr (3 - 1)
⇒ 3 * (2 * facr (2 - 1))
let rec facr n =
    if n = 0
    then 1
    else n * facr (n - 1)

facr 3
⇒ 3 * facr (3 - 1)
⇒ 3 * (2 * facr (2 - 1))
⇒ 3 * (2 * (1 * facr (1 - 1)))
Stack frames and continuations

let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

facr 3
⇒ 3 * facr (3 - 1)
⇒ 3 * (2 * facr (2 - 1))
⇒ 3 * (2 * (1 * facr (1 - 1)))
⇒ 3 * (2 * (1 * 1))
⇒ 3 * 2
⇒ 6

facr 0 : 1
facr 1 : 1 * □
facr 2 : 2 * □
facr 3 : 3 * □
main: print □
globals
let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

facr 3
⇒ 3 * facr (3 - 1)
⇒ 3 * (2 * facr (2 - 1))
⇒ 3 * (2 * (1 * facr (1 - 1)))
⇒ 3 * (2 * (1 * 1))
⇒ 3 * (2 * 1)
let rec facr n =
    if n = 0
    then 1
    else n * facr (n - 1)

facr 3
⇒ 3 * facr (3 - 1)
⇒ 3 * (2 * facr (2 - 1))
⇒ 3 * (2 * (1 * facr (1 - 1)))
⇒ 3 * (2 * (1 * 1))
⇒ 3 * (2 * 1)
⇒ 3 * 2
let rec facr n =
    if n = 0
    then 1
    else n * facr (n - 1)

facr 3
⇒ 3 * facr (3 - 1)
⇒ 3 * (2 * facr (2 - 1))
⇒ 3 * (2 * (1 * facr (1 - 1)))
⇒ 3 * (2 * (1 * 1))
⇒ 3 * (2 * 1)
⇒ 3 * 2
⇒ 6
let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

  facr 3
  \[\Rightarrow 3 \times \text{facr} (3 - 1)\]
  \[\Rightarrow 3 \times (2 \times \text{facr} (2 - 1))\]
  \[\Rightarrow 3 \times (2 \times (1 \times \text{facr} (1 - 1)))\]
  \[\Rightarrow 3 \times (2 \times (1 \times 1))\]
  \[\Rightarrow 3 \times (2 \times 1)\]
  \[\Rightarrow 3 \times 2\]
  \[\Rightarrow 6\]

  The continuation is the “rest of the computation”.

main: print 6

The continuation is the “rest of the computation”.
What is a continuation?

Metaphors for “the rest of the computation”

- The waiting stack, upside down
What is a continuation?

Metaphors for “the rest of the computation”

- The waiting stack, upside down
- Functional GOTO labels
What is a continuation?

Metaphors for “the rest of the computation”

- The waiting stack, upside down
- Functional GOTO labels
- The rest of the program, with a “hole”

Continuation passing style (CPS) lets us use continuations in most languages
Uses of continuations

- A function in CPS can sometimes be rewritten to use an accumulating parameter, saving memory.
Uses of continuations

- A function in CPS can sometimes be rewritten to use an accumulating parameter, saving memory
- A function in CPS can sometimes stop the computation early, saving time
Uses of continuations

- A function in CPS can sometimes be rewritten to use an accumulating parameter, saving memory
- A function in CPS can sometimes stop the computation early, saving time
- An interpreter in CPS can model exceptions and exception handling try-catch
- Continuations can implement expressions with multiple results, as in Icon and Prolog
- Continuation-thinking helps on-the-fly optimization in the micro-C compiler (next lecture)
- Continuations can be used to structure web dialogs
- Continuations have many other more magical uses
Uses of continuations

- A function in CPS can sometimes be rewritten to use an accumulating parameter, saving memory
- A function in CPS can sometimes stop the computation early, saving time
- An interpreter in CPS can model exceptions and exception handling try-catch
- Continuations can implement expressions with multiple results, as in Icon and Prolog
Uses of continuations

- A function in CPS can sometimes be rewritten to use an accumulating parameter, saving memory
- A function in CPS can sometimes stop the computation early, saving time
- An interpreter in CPS can model exceptions and exception handling try-catch
- Continuations can implement expressions with multiple results, as in Icon and Prolog
- Continuation-thinking helps on-the-fly optimization in the micro-C compiler (next lecture);
Uses of continuations

- A function in CPS can sometimes be rewritten to use an accumulating parameter, saving memory
- A function in CPS can sometimes stop the computation early, saving time
- An interpreter in CPS can model exceptions and exception handling try-catch
- Continuations can implement expressions with multiple results, as in Icon and Prolog
- Continuation-thinking helps on-the-fly optimization in the micro-C compiler (next lecture)
- Continuations can be used to structure web dialogs
Uses of continuations

- A function in CPS can sometimes be rewritten to use an accumulating parameter, saving memory
- A function in CPS can sometimes stop the computation early, saving time
- An interpreter in CPS can model exceptions and exception handling try-catch
- Continuations can implement expressions with multiple results, as in Icon and Prolog
- Continuation-thinking helps on-the-fly optimization in the micro-C compiler (next lecture);
- Continuations can be used to structure web dialogs
- Continuations have many other more magical uses
Overview

CONTINUATIONS AND CONTINUATION-PASSING STYLE
Stack frames and continuations
Continuation-passing style
Tail recursion and iteration
CPS in Java

IMPLEMENTING EXCEPTIONS
Throwing exceptions
Handling exceptions

MICRO-ICON
Micro-Icon introduction
Micro-Icon interpreter
Continuation-Passing Style (CPS)

let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

▶ Each function gets a continuation argument \( k \)

let rec facc n \( k \) =
  if n = 0
  then \( k \) 1
  else facc (n - 1) (fun v -> \( k \) (n * v))
Continuation-Passing Style (CPS)

let rec facr n = 
    if n = 0 
    then 1 
    else n * facr (n - 1) 

▶ Each function gets a continuation argument \( k \)
▶ Do not return \( res \) - instead call \( k \ res \)

let rec facc n k = 
    if n = 0 
    then \( k \ 1 \) 
    else facc (n - 1) (fun v -> \( k \ (n * v) \))
Continuation-Passing Style (CPS)

let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

- Each function gets a continuation argument $k$
- Do not return $res$ - instead call $k res$
- $k$ takes care of the result

let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))
let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)
let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

let rec facc n k =
  if n = 0
  then ???
  else ???
Deriving a CPS facr

let rec facr n =
    if n = 0
    then 1
    else n * facr (n - 1)

let rec facc n k =
    if n = 0
    then k 1
    else ???

- Add continuation argument
- If n = 0, send 1 to the continuation
Deriving a CPS facr

let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> v)

- Add continuation argument
- If $n = 0$, send 1 to the continuation
- Otherwise call recursively, with new continuation
Deriving a CPS facr

let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

- Add continuation argument
- If \( n = 0 \), send 1 to the continuation
- Otherwise call recursively, with new continuation
- Represent continuation as a function
Deriving a CPS facr

let rec facr n =  
    if n = 0  
    then 1  
    else n * facr (n - 1) 

let rec facc n k =  
    if n = 0  
    then k 1  
    else facc (n - 1) (fun v -> k (n * v))

- Add continuation argument
- If \( n = 0 \), send 1 to the continuation
- Otherwise call recursively, with new continuation
- Represent continuation as a function

\( \text{facr} n = \text{facc} n \ (\text{fun} \ v \rightarrow \ v) \)
Evaluating facc

let rec facc n k =
    if n = 0
    then k 1
    else facc (n - 1) (fun v -> k (n * v))

let id x = x

  facc 3 id
let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

  facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
⇒ facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
⇒ facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
⇒ facc 0 (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u))
⇒ (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u)) 1
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * 1)
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) 1
⇒ (fun v -> id (3 * v)) (2 * 1)
⇒ (fun v -> id (3 * v)) 2
⇒ id (3 * 2)
⇒ id 6
⇒ 6
Evaluating facc

```ocaml
let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
⇒ facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
⇒ facc 0 (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u))
⇒ (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u)) 1
```

```'
main: print □
globals
```
Evaluating facc

let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
⇒ facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
⇒ facc 0 (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u))
⇒ (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u)) 1
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * 1)
Evaluating facc

let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
⇒ facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
⇒ facc 0 (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u))
⇒ (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u)) 1
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * 1)
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) 1
⇒ id (3 * 2)
⇒ id 6
⇒ 6
Evaluating facc

let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
⇒ facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
⇒ facc 0 (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u))
⇒ (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u)) 1
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * 1)
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) 1
⇒ (fun v -> id (3 * v)) (2 * 1)
let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
⇒ facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
⇒ facc 0 (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u))
⇒ (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u)) 1
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * 1)
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) 1
⇒ (fun v -> id (3 * v)) (2 * 1)
⇒ (fun v -> id (3 * v)) 2
⇒ id (3 * 2)
⇒ id 6
⇒ 6
Evaluating facc

let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
⇒ facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
⇒ facc 0 (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u))
⇒ (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u)) 1
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * 1)
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) 1
⇒ (fun v -> id (3 * v)) (2 * 1)
⇒ (fun v -> id (3 * v)) 2
⇒ id (3 * 2)

main: print □
globals
Evaluating facc

let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
⇒ facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
⇒ facc 0 (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u))
⇒ (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u)) 1
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * 1)
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) 1
⇒ (fun v -> id (3 * v)) (2 * 1)
⇒ (fun v -> id (3 * v)) 2
⇒ id (3 * 2)
⇒ id 6

main: print □
globals

IT UNIVERSITY OF COPENHAGEN
Evaluating facc

let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

let id x = x

facc 3 id
⇒ facc 2 (fun v -> id (3 * v))
⇒ facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
⇒ facc 0 (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u))
⇒ (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u)) 1
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * 1)
⇒ (fun w -> (fun v -> id (3 * v)) (2 * w)) 1
⇒ (fun v -> id (3 * v)) (2 * 1)
⇒ (fun v -> id (3 * v)) 2
⇒ id (3 * 2)
⇒ id 6
⇒ 6

main: print 6
Exercise

Convert the following function to CPS.

```ml
let rec prod xs =
match xs with
| []      -> 1
| x :: xr -> x * prod xr
```

Hint: start with

```ml
let rec prodc xs k =
match xs with
| []      -> ???
| x :: xr -> ???
```

IT UNIVERSITY OF COPENHAGEN
Break
Overview

CONTINUATIONS AND CONTINUATION-PASSING STYLE
- Stack frames and continuations
- Continuation-passing style
- Tail recursion and iteration
- CPS in Java

IMPLEMENTING EXCEPTIONS
- Throwing exceptions
- Handling exceptions

MICRO-ICON
- Micro-Icon introduction
- Micro-Icon interpreter
Rewrite $\text{facr}$ with accumulator:

$$
\begin{align*}
\text{let rec faci } n \ r &= \\
\text{if } n = 0 &\quad \text{then } r \\
\text{else faci } (n - 1) \ (r \times n) \\
\end{align*}
$$

$\text{faci } 3 \ 1$

$\text{faci } n \ r = r \times \text{facr } n$

$\text{facr } n = \text{faci } n \ 1$
Tail recursion and iteration

Rewrite \( \text{facr} \) with accumulator:

\[
\text{let rec faci n r = } \\
\quad \text{if } n = 0 \text{ then } r \\
\quad \text{else faci (n - 1) (r * n)} \\
\]

\[
\text{faci 3 1} \\
\implies \text{faci 2 3} \\
\]

\[
\text{faci n r = r * facr n} \\
\text{facr n = faci n 1} \\
\]
Rewrite $\text{facr}$ with accumulator:

\[
\text{let rec faci } \ n \ r = \\
\text{ if } n = 0 \text{ then } r \\
\text{ else faci } (n - 1) \ (r \times n)
\]

\[
\text{faci } 3 \ 1 \\
\Rightarrow \text{ faci } 2 \ 3 \\
\Rightarrow \text{ faci } 1 \ 6
\]

\[
\text{faci } n \ r = r \times \text{facr } n \\
\text{facr } n = \text{faci } n \ 1
\]
Rewrite $\text{facr}$ with accumulator:

```ocaml
defac n r = 
  if n = 0 
  then r 
  else defac (n - 1) (r * n)

defac 3 1
⇒ defac 2 3
⇒ defac 1 6
⇒ defac 0 6
```

$\text{defac} n r = r * \text{facr} n$

$\text{facr} n = \text{defac} n 1$
Rewrite $\text{facr}$ with accumulator:

\[
\begin{align*}
\text{let rec faci } n & \quad r = \\
& \quad \text{if } n = 0 \\
& \quad \text{then } r \\
& \quad \text{else faci } (n - 1) \ (r \ast n)
\end{align*}
\]

\[
\begin{align*}
\text{faci } 3 & \quad 1 \\
\Rightarrow & \quad \text{faci } 2 \quad 3 \\
\Rightarrow & \quad \text{faci } 1 \quad 6 \\
\Rightarrow & \quad \text{faci } 0 \quad 6 \\
\Rightarrow & \quad 6
\end{align*}
\]
Continuations vs accumulating parameters

▶ Which of \texttt{facr n}, \texttt{facc n k}, \texttt{faci n r} are tail-recursive?

\[ k \text{ is always } \texttt{fun u -> r * u}. \]
\[ \text{Proof:} \]
\[ \text{At the top call, } k = \texttt{id} = \texttt{fun u -> u} = \texttt{fun u -> 1 * u}. \]
\[ \text{If an argument } k \text{ has form } \texttt{k = fun u -> r * u}, \text{ then the new continuation is:} \]
\[ \texttt{fun v -> k (n * v)} = \texttt{fun v -> (fun u -> r * u) (n * v)} = \texttt{fun v -> r * (n * v)} = \texttt{fun v -> (r * n) * v}. \]

Thus, \[ r \] is a simple representation of \[ k \]
Continuations vs accumulating parameters

- Which of facr n, facc n k, faci n r are tail-recursive?
- What is the relationship between k and r?

Proof:
- At the top call, \( k = \text{id} = \lambda u . u = \lambda u . 1 \cdot u \)
- If an argument \( k = \lambda u . r \cdot u \), then the new continuation is:
  \[
  \lambda v . k(n \cdot v) = \lambda v . (\lambda u . r \cdot u)(n \cdot v) = \lambda v . r \cdot (n \cdot v) = \lambda v . (r \cdot n) \cdot v
  \]
- Thus, \( r \) is a simple representation of \( k \)

All functions can be made tail recursive - but only some continuations can be represented simply.
Continuations vs accumulating parameters

- Which of facr n, facc n k, faci n r are tail-recursive?
- What is the relationship between k and r?
- k is always fun u -> r * u. Proof:

  At the top call, k = id = fun u -> u = fun u -> 1 * u

  If an argument k has form k = fun u -> r * u, then the new continuation is:
  fun v -> k (n * v) = fun v -> (fun u -> r * u) (n * v) = fun v -> r * (n * v) = fun v -> (r * n) * v

  Thus, r is a simple representation of k

  All functions can be made tail recursive — but only some continuations can be represented simply.
Which of facr n, facc n k, faci n r are tail-recursive?

What is the relationship between k and r?

k is always fun u -> r * u. Proof:

At the top call, k = id = fun u -> u = fun u -> 1 * u
Continuations vs accumulating parameters

- Which of \texttt{facr n}, \texttt{facc n k}, \texttt{faci n r} are tail-recursive?
- What is the relationship between \(k\) and \(r\)?
- \(k\) is always \texttt{fun u -> r * u}. Proof:
  - At the top call, \(k = \text{id} = \text{fun u -> u = fun u -> 1 * u}\)
  - If an argument \(k\) has form \(k = \text{fun u -> r * u}\), then the new continuation is:
    \[
    \text{fun v -> k (n * v)}
    \]
Which of facr n, facc n k, faci n r are tail-recursive?

What is the relationship between k and r?

k is always fun u -> r * u. Proof:

At the top call, k = id = fun u -> u = fun u -> 1 * u

If an argument k has form k = fun u -> r * u, then the new continuation is:

fun v -> k (n * v)
= fun v -> (fun u -> r * u) (n * v)
Continuations vs accumulating parameters

- Which of facr n, facc n k, faci n r are tail-recursive?
- What is the relationship between k and r?
- k is always fun u -> r * u. Proof:
  - At the top call, k = id = fun u -> u = fun u -> 1 * u
  - If an argument k has form k = fun u -> r * u, then the new continuation is:
    - fun v -> k (n * v)
    = fun v -> (fun u -> r * u) (n * v)
    = fun v -> r * (n * v)
Which of \texttt{facr} \(n\), \texttt{facc} \(n\) \(k\), \texttt{faci} \(n\) \(r\) are tail-recursive?

What is the relationship between \(k\) and \(r\)?

\(k\) is always \texttt{fun} \(u \rightarrow r \ast u\). Proof:

\begin{itemize}
  \item At the top call, \(k = \text{id} = \text{fun} \ u \rightarrow u = \text{fun} \ u \rightarrow 1 \ast u\)
  \item If an argument \(k\) has form \(k = \text{fun} \ u \rightarrow r \ast u\), then the new continuation is:
    \[
    \text{fun} \ v \rightarrow k (n \ast v) \\
    = \text{fun} \ v \rightarrow (\text{fun} \ u \rightarrow r \ast u) (n \ast v) \\
    = \text{fun} \ v \rightarrow r \ast (n \ast v) \\
    = \text{fun} \ v \rightarrow (r \ast n) \ast v
    \]
\end{itemize}
Which of \( \text{facr } n, \text{facc } n \ k, \text{faci } n \ r \) are tail-recursive?

What is the relationship between \( k \) and \( r \)?

\( k \) is always \( \text{fun } u \rightarrow r \ast u \). Proof:

- At the top call, \( k = \text{id} = \text{fun } u \rightarrow u = \text{fun } u \rightarrow 1 \ast u \)
- If an argument \( k \) has form \( k = \text{fun } u \rightarrow r \ast u \), then the new continuation is:
  \[
  \text{fun } v \rightarrow k(n \ast v) \\
  = \text{fun } v \rightarrow (\text{fun } u \rightarrow r \ast u)(n \ast v) \\
  = \text{fun } v \rightarrow r \ast (n \ast v) \\
  = \text{fun } v \rightarrow (r \ast n) \ast v
  \]
- Thus, \( r \) is a simple representation of \( k \)
Continuations vs accumulating parameters

- Which of `facr n`, `facc n k`, `faci n r` are tail-recursive?
- What is the relationship between `k` and `r`?
- `k` is always `fun u -> r * u`. Proof:
  - At the top call, `k = id = fun u -> u = fun u -> 1 * u`
  - If an argument `k` has form `k = fun u -> r * u`, then the new continuation is:
    
    ```
    fun v -> k (n * v)  
    = fun v -> (fun u -> r * u) (n * v)  
    = fun v -> r * (n * v)  
    = fun v -> (r * n) * v  
    ```
- Thus, `r` is a simple representation of `k`
- All functions can be made tail recursive - but only some continuations can be represented simply
CONTINUATIONS AND CONTINUATION-PASSING STYLE
- Stack frames and continuations
- Continuation-passing style
- Tail recursion and iteration
- CPS in Java

IMPLEMENTING EXCEPTIONS
- Throwing exceptions
- Handling exceptions

MICRO-ICON
- Micro-Icon introduction
- Micro-Icon interpreter
CONTINUATIONS

/* Representing functions int -> int */
interface Cont {
    int k(int v);
}
CONTINUATIONS

/* Representing functions int -> int */
interface Cont {
    int k(int v);
}

CPS FACTORIAL

static int facc(final int n, final Cont cont) {
    if (n == 0)
        return cont.k(1);
    else
        return facc(n - 1,
                    new Cont() {
                        public int k(int v) {
                            return cont.k(n * v);
                        }
                    });
}
CONTINUATIONS

/* Representing functions int -> int */
interface Cont {
    int k(int v);
}

CPS FACTORIAL

static int facc(final int n, final Cont cont) {
    if (n == 0)
        return cont.k(1);
    else
        return facc(n - 1,
                    new Cont() {
                        public int k(int v) {
                            return cont.k(n * v);
                        }
                    });
}
CPS in Java

/* Representing functions int -> int */

interface Cont {
    int k(int v);
}

CPS FACTORIAL

static int facc(final int n, final Cont cont) {
    if (n == 0)
        return cont.k(1);
    else
        return facc(n - 1, new Cont() {
            public int k(int v) {
                return cont.k(n * v);
            }
        });
}
Why CPS?

- In normal code, continuations are implicit:
  - Surrounding expressions
  - Next statement
  - Activation records on the stack

- Making the continuation explicit:
  - We can ignore it, "avoiding returning"
  - We can have two continuations, choosing "how to return"
  - Ignoring the continuation = throwing an exception
  - Choosing a continuation is good for:
    - handling exceptions, and
    - producing multiple results from an expression
Why CPS?

- In normal code, continuations are implicit:
  - Surrounding expressions
  - Next statement
  - Activation records on the stack
- Making the continuation explicit:
  - We can ignore it, “avoiding returning”
  - We can have two continuations, choosing “how to return”
Why CPS?

- In normal code, continuations are implicit:
  - Surrounding expressions
  - Next statement
  - Activation records on the stack

- Making the continuation explicit:
  - We can ignore it, “avoiding returning”
  - We can have two continuations, choosing “how to return”

- Ignoring the continuation = throwing an exception
Why CPS?

► In normal code, continuations are implicit:
  ▶ Surrounding expressions
  ▶ Next statement
  ▶ Activation records on the stack

► Making the continuation explicit:
  ▶ We can ignore it, “avoiding returning”
  ▶ We can have two continuations, choosing “how to return”

► Ignoring the continuation = throwing an exception

► Choosing a continuation is good for:
  ▶ handling exceptions, and
  ▶ producing multiple results from an expression
CONTINUATIONS AND CONTINUATION-PASSING STYLE
Stack frames and continuations
Continuation-passing style
Tail recursion and iteration
CPS in Java

IMPLEMENTING EXCEPTIONS
Throwing exceptions
Handling exceptions

MICRO-ICON
Micro-Icon introduction
Micro-Icon interpreter
A simple functional language with exceptions

type expr =
| ... |
| Raise of exn               // raise exn |
| TryWith of expr * exn * expr // try e1 with exn -> e2 |
A simple functional language with exceptions

type expr =
| ...
| Raise of exn // raise exn
| TryWith of expr * exn * expr // try e1 with exn -> e2

Evaluation now yields an integer or fails with an error message:

type answer =
| Result of int
| Abort of string

let rec coEval1 e env (cont : int -> answer) : answer = ...
Overview

CONTINUATIONS AND CONTINUATION-PASSING STYLE
Stack frames and continuations
Continuation-passing style
Tail recursion and iteration
CPS in Java

IMPLEMENTING EXCEPTIONS
Throwing exceptions
Handling exceptions

MICRO-ICON
Micro-Icon introduction
Micro-Icon interpreter
let rec coEval1 e env (cont : int -> answer) : answer =
  match e with
  | CstI i -> cont i
  | Var x ->
    match lookup env x with
    | Int i -> cont i
    | _ -> Abort "coEval1 Var"
  | Prim(ope, e1, e2) ->
    coEval1 e1 env (fun i1 ->
      coEval1 e2 env (fun i2 ->
        match ope with
        | '*' -> cont (i1 * i2)
        | '+' -> cont (i1 + i2)
        | ... ))
  | Raise (Exn s) -> Abort s
let rec coEval1 e env (cont : int -> answer) : answer =
  match e with
  | CstI i -> cont i
  | Var x ->
    match lookup env x with
    | Int i -> cont i
    | _     -> Abort "coEval1 Var"
  | Prim(ope, e1, e2) ->
    coEval1 e1 env (fun i1 ->
      coEval1 e2 env (fun i2 ->
        match ope with
        | "*" -> cont (i1 * i2)
        | "+" -> cont (i1 + i2)
        | ... ))
  | Raise (Exn s) -> Abort s
let rec coEval1 e env (cont : int -> answer) : answer =
  match e with
  | CstI i -> cont i
  | Var x  ->
    match lookup env x with
    | Int i -> cont i
    | _    -> Abort "coEval1 Var"
  | Prim(ope, e1, e2) ->
    coEval1 e1 env
    (fun i1 ->
      coEval1 e2 env
      (fun i2 ->
        match ope with
        | "*" -> cont (i1 * i2)
        | "+" -> cont (i1 + i2)
        | ... )))
let rec coEval1 e env (cont : int -> answer) : answer =
    match e with
    | CstI i -> cont i
    | Var x ->
        match lookup env x with
        | Int i -> cont i
        | _ -> Abort "coEval1 Var"
    | Prim(ope, e1, e2) ->
        coEval1 e1 env
        (fun i1 ->
            coEval1 e2 env
            (fun i2 ->
                match ope with
                | "*" -> cont (i1 * i2)
                | "+" -> cont (i1 + i2)
                | ... ))
    | Raise (Exn s) -> Abort s
let rec coEval1 e env (cont : int -> answer) : answer =
  match e with
  | ... | If(e1, e2, e3) ->
    coEval1 e1 env
    (fun b -> if b <> 0 then
        coEval1 e2 env cont
        else
        coEval1 e3 env cont)
  | ... | ...
let rec coEval1 e env (cont : int -> answer) : answer =
match e with
| ... |
| If(e1, e2, e3) ->
  coEval1 e1 env
  (fun b -> if b <> 0 then
    coEval1 e2 env cont
  else
    coEval1 e3 env cont)
| ... |
Overview

CONTINUATIONS AND CONTINUATION-PASSING STYLE
- Stack frames and continuations
- Continuation-passing style
- Tail recursion and iteration
- CPS in Java

IMPLEMENTING EXCEPTIONS
- Throwing exceptions
- Handling exceptions

MICRO-ICON
- Micro-Icon introduction
- Micro-Icon interpreter
Interpreter for handling exceptions

- Add an error continuation to the interpreter:
  econt : exn -> answer
Interpreter for handling exceptions

- Add an error continuation to the interpreter:
  \[
  \text{econt} : \ exn \rightarrow \ \text{answer}
  \]

- To throw exception, call error continuation instead of normal continuation
Interpreter for handling exceptions

- Add an error continuation to the interpreter:
  \[\text{econt} : \text{exn} \rightarrow \text{answer}\]

- To throw exception, call error continuation instead of normal continuation

- The error continuation decides whether or not to handle the exception
Non-exception evaluation is as before:

```plaintext
let rec coEval2 e env (cont : int -> answer) (econt : exn -> answer) : answer =
  match e with
  | CstI i -> cont i
  | If(e1, e2, e3) ->
    coEval2 e1 env (fun b ->
      if b <> 0 then
        coEval2 e2 env cont econt
      else
        coEval2 e3 env cont econt)
    econt
  | ...
```
...  
| Raise exn -> econt exn  
| TryWith (e1, exn, e2) ->  
  let econt1 thrown =  
    if thrown = exn  
      then coEval2 e2 env cont econt  
      else econt thrown  
  in coEval2 e1 env cont econt1
Interpreter for throwing and handling exceptions

...  
| Raise exn -> econt exn  
| TryWith (e1, exn, e2) ->  
  let econt1 thrown =  
    if thrown = exn  
    then coEval2 e2 env cont econt  
    else econt thrown  
  in coEval2 e1 env cont econt1

Throw the exception to the current error handler
...  
| Raise exn -> econt exn  
| TryWith (e1, exn, e2) ->  
  let econt1 thrown =  
   if thrown = exn  
   then coEval2 e2 env cont econt  
   else econt thrown  
  in coEval2 e1 env cont econt1

Exception handlers make new error continuations
Interpreter for throwing and handling exceptions

... 
| Raise exn -> econt exn 
| TryWith (e1, exn, e2) -> 
  let econt1 thrown = 
    if thrown = exn 
    then coEval2 e2 env cont econt 
    else econt thrown 
  in coEval2 e1 env cont econt1

If the new error continuation gets a matching error, call handler
... | Raise exn -> econt exn
| TryWith (e1, exn, e2) ->
  let econt1 thrown =
    if thrown = exn
    then coEval2 e2 env cont econt
    else econt thrown
  in coEval2 e1 env cont econt1

If the error doesn’t match, pass it up to next error handler
Break
CONTINUATIONS AND CONTINUATION-PASSING STYLE
- Stack frames and continuations
- Continuation-passing style
- Tail recursion and iteration
- CPS in Java

IMPLEMENTING EXCEPTIONS
- Throwing exceptions
- Handling exceptions

MICRO-ICON
- Micro-Icon introduction
- Micro-Icon interpreter
Expressions giving multiple results; the Icon language

<table>
<thead>
<tr>
<th>Expression</th>
<th>Results</th>
<th>Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>Constant</td>
</tr>
<tr>
<td>write 5</td>
<td>5</td>
<td>5</td>
<td>Constant, side effect</td>
</tr>
<tr>
<td>(1 to 3)</td>
<td>1 2 3</td>
<td></td>
<td>Range, 3 results</td>
</tr>
<tr>
<td>write (1 to 3)</td>
<td>1 2 3</td>
<td>1</td>
<td>Side effect</td>
</tr>
<tr>
<td>every (write (1 to 3))</td>
<td>0</td>
<td>1 2 3</td>
<td>Force all results</td>
</tr>
<tr>
<td>(1 to 0)</td>
<td></td>
<td></td>
<td>Empty range, no res.</td>
</tr>
<tr>
<td>&amp;fail</td>
<td></td>
<td></td>
<td>No results</td>
</tr>
<tr>
<td>(1 to 3)+(4 to 6)</td>
<td>5 6 7 6 7 8 7 8 9</td>
<td>All combinations</td>
<td></td>
</tr>
</tbody>
</table>
Expressions giving multiple results; the Icon language

<table>
<thead>
<tr>
<th>Expression</th>
<th>Results</th>
<th>Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 &lt; 4</td>
<td>4</td>
<td></td>
<td>Comparison succeeds</td>
</tr>
<tr>
<td>4 &lt; 3</td>
<td></td>
<td></td>
<td>Comparison fails</td>
</tr>
<tr>
<td>3 &lt; (1 to 5)</td>
<td>4 5</td>
<td></td>
<td>Succeeds twice</td>
</tr>
<tr>
<td>(1 to 3)</td>
<td>(4 to 6)</td>
<td>1 2 3 4 5 6</td>
<td>Each left, each right</td>
</tr>
<tr>
<td>(1 to 3) &amp; (4 to 6)</td>
<td>4 5 6 4 5 6</td>
<td></td>
<td>Each right for each left</td>
</tr>
<tr>
<td>(1 to 3) ; (4 to 6)</td>
<td>4 5 6</td>
<td></td>
<td>No backtracking to left</td>
</tr>
</tbody>
</table>
Exercise

What does the following expression do?

- every (write ((1 | 7) * (2 | 3)))

Write Icon expressions to print the following:

- 2 4 6 8 10
- 2 4 6 7 8
Break
Overview

CONTINUATIONS AND CONTINUATION-PASSING STYLE
Stack frames and continuations
Continuation-passing style
Tail recursion and iteration
CPS in Java

IMPLEMENTING EXCEPTIONS
Throwing exceptions
Handling exceptions

MICRO-ICON
Micro-Icon introduction
Micro-Icon interpreter
The interpreter takes two continuations:

**FAILURE CONTINUATION** :

\[
\text{econt : unit} \rightarrow \text{answer}
\]
called when there are no (more) results

**SUCCESS CONTINUATION** :

\[
\text{cont : value} \rightarrow \text{econt} \rightarrow \text{answer}
\]
called when there is one (more) result
Micro-Icon interpreter

The interpreter takes two continuations:

**FAILURE CONTINUATION**:

\[ \text{econt} : \text{unit} \rightarrow \text{answer} \]

called when there are no (more) results

**SUCCESS CONTINUATION**:

\[ \text{cont} : \text{value} \rightarrow \text{econt} \rightarrow \text{answer} \]

called when there is one (more) result

The `econt` argument to `cont` can be called by `cont` to ask for more results:

```ml
let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | CstI i -> cont (Int i) econt
  | ...   |
  | Fail -> econt ()
```


let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | CstI i -> cont (Int i) econt
  | CstS s -> cont (Str s) econt
  | Prim(ope, e1, e2) ->
    eval e1 (fun v1 -> fun econt1 ->
      eval e2 (fun v2 -> fun econt2 ->
        match (ope, v1, v2) with
        | ("+", Int i1, Int i2) ->
          cont (Int(i1 + i2)) econt2
        | ("*", Int i1, Int i2) ->
          cont (Int(i1 * i2)) econt2
        | (<, Int i1, Int i2) ->
          if i1 < i2 then
            cont (Int i2) econt2
          else
            econt2 ()
        | _ -> Str "unknown prim2")
      )
    )
  | ...

let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | CstI i -> cont (Int i) econt
  | CstS s -> cont (Str s) econt
  | Prim(ope, e1, e2) ->
    eval e1 (fun v1 ->
      fun econt1 ->
        eval e2 (fun v2 ->
          fun econt2 ->
            match (ope, v1, v2) with
            | (+, Int i1, Int i2) ->
              cont (Int (i1 + i2)) econt2
            | (*, Int i1, Int i2) ->
              cont (Int (i1 * i2)) econt2
            | (<, Int i1, Int i2) ->
              if i1 < i2 then
                cont (Int i2) econt2
              else
                econt2 ()
            | _ -> Str "unknown prim2")
          econt1)
        econt)
      econt
    ...
let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | CstI i -> cont (Int i) econt
  | CstS s -> cont (Str s) econt
  | Prim(ope, e1, e2) ->
    eval e1 (fun v1 -> fun econt1 ->
      eval e2 (fun v2 -> fun econt2 ->
        match (ope, v1, v2) with
        | ("+", Int i1, Int i2) ->
          cont (Int(i1 + i2)) econt2
        | ("*", Int i1, Int i2) ->
          cont (Int(i1 * i2)) econt2
        | ("<", Int i1, Int i2) ->
          if i1 < i2 then
            cont (Int i2) econt2
          else
            econt2 ()
        | _ -> Str "unknown prim2")
      econt1)
    econt
  | ...
let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | CstI i -> cont (Int i) econt
  | CstS s -> cont (Str s) econt
  | Prim(ope, e1, e2) ->
      eval e1 (fun v1 -> fun econt1 ->
        eval e2 (fun v2 -> fun econt2 ->
          match (ope, v1, v2) with
          | ("+", Int i1, Int i2) ->
            cont (Int(i1 + i2)) econt2
          | ("*", Int i1, Int i2) ->
            cont (Int(i1 * i2)) econt2
          | ("<", Int i1, Int i2) ->
            if i1 < i2 then
              cont (Int i2) econt2
            else
              econt2 ()
          | _ -> Str "unknown prim2")
        econt1)
      econt
    | ...
let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | CstI i -> cont (Int i) econt
  | CstS s -> cont (Str s) econt
  | Prim(ope, e1, e2) ->
    eval e1 (fun v1 -> fun econt1 ->
      eval e2 (fun v2 -> fun econt2 ->
        match (ope, v1, v2) with
        | ("+", Int i1, Int i2) ->
          cont (Int(i1 + i2)) econt2
        | ("*", Int i1, Int i2) ->
          cont (Int(i1 * i2)) econt2
        | ("<", Int i1, Int i2) ->
          if i1 < i2 then
            cont (Int i2) econt2
          else
            econt2 ()
        | _ -> Str "unknown prim2")
      econt1)
    econt
  | ...

Send results to outer continuation, using inner error handler
let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | CstI i -> cont (Int i) econt
  | CstS s -> cont (Str s) econt
  | Prim(ope, e1, e2) ->
    eval e1 (fun v1 -> fun econt1 ->
      eval e2 (fun v2 -> fun econt2 ->
        match (ope, v1, v2) with
        | ("+", Int i1, Int i2) ->
          cont (Int(i1 + i2)) econt2
        | ("*", Int i1, Int i2) ->
          cont (Int(i1 * i2)) econt2
        | ("<", Int i1, Int i2) ->
          if i1 < i2 then
            cont (Int i2) econt2
          else
            econt2 ()
        | _ -> Str "unknown prim2")
      econt2)
    econt1)
  | ...
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | CstI i -> cont (Int i) econt
    | CstS s -> cont (Str s) econt
    | Prim(ope, e1, e2) ->
      eval e1 (fun v1 -> fun econt1 ->
        eval e2 (fun v2 -> fun econt2 ->
          match (ope, v1, v2) with
          | ("+", Int i1, Int i2) ->
            cont (Int(i1 + i2)) econt2
          | ("*", Int i1, Int i2) ->
            cont (Int(i1 * i2)) econt2
          | ("<", Int i1, Int i2) ->
            if i1 < i2 then
              cont (Int i2) econt2
            else
              econt2 ()
          | _ -> Str "unknown prim2")
        econt1)
      econt
    | ...

For real errors, stop program without using continuations
let rec eval (e : expr) (cont : cont) (econt : econt) = 
    match e with 
    | FromTo(i1, i2) -> 
      let rec loop i = 
        if i <= i2 then 
          cont (Int i) (fun () -> loop (i+1)) 
        else 
          econt () 
      in loop i1 
    | Write e -> 
      eval e (fun v -> 
        fun econt1 -> (write v; cont v econt1)) 
      econt 
    | If(e1, e2, e3) -> 
      eval e1 (fun _ -> fun _ -> eval e2 cont econt) 
      (fun () -> eval e3 cont econt) 
    | ...
let rec eval (e : expr) (cont : cont) (econt : econt) =
match e with
    | ... |
    | FromTo(i1, i2) ->
        let rec loop i =
            let rec loop i =
                if i <= i2 then
                    if i <= i2 then
                        cont (Int i) (fun () -> loop (i+1))
                    else
                        econt ()
                in loop i1
            in loop i1
        in loop i1
    | Write e ->
        eval e (fun v ->
            fun econt1 -> (write v; cont v econt1))
            econt
    | If(e1, e2, e3) ->
        eval e1 (fun _ -> fun _ -> eval e2 cont econt)
            (fun () -> eval e3 cont econt)
    | ... |

Handle 1 to 3
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | ... |
    | FromTo(i1, i2) ->
      let rec loop i =
        if i <= i2 then
          cont (Int i) (fun () -> loop (i+1))
        else
          econt ()
      in loop i1
    | Write e ->
      eval e (fun v -> fun econt1 -> (write v; cont v econt1)) econt
    | If(e1, e2, e3) ->
      eval e1 (fun _ -> fun _ -> eval e2 cont econt) (fun () -> eval e3 cont econt)
    | ...

While values are left, send them to the success continuation
let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | ...  
  | FromTo(i1, i2) ->
    let rec loop i =
      if i <= i2 then
        cont (Int i) (fun () -> loop (i+1))
      else
        econt ()
    in loop i1
  | Write e ->
    eval e (fun v ->
      fun econt1 -> (write v; cont v econt1)) econt
  | If(e1, e2, e3) ->
    eval e1 (fun _ -> fun _ -> eval e2 cont econt)
    (fun () -> eval e3 cont econt)
  | ...

cont gets the next loop iteration in case of failure
let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | ... 
  | FromTo(i1, i2) ->
    let rec loop i =
      if i <= i2 then
        cont (Int i) (fun () -> loop (i+1))
      else
        econt ()
    in loop i1
  | Write e ->
    eval e (fun v ->
      fun econt1 -> (write v; cont v econt1)) econt
  | If(e1, e2, e3) ->
    eval e1 (fun _ ->
      fun _ -> eval e2 cont econt)
                      (fun () -> eval e3 cont econt)
  | ...

When done looping, go back to previous failure continuation
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | ... |
    | FromTo(i1, i2) ->
      let rec loop i =
        if i <= i2 then
          cont (Int i) (fun () -> loop (i+1))
        else
          econt ()
      in loop i1
    | Write e ->
      eval e (fun v ->
        fun econt1 -> (write v; cont v econt1))
      econt

Eval e, then write it and return it
let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | ...
  | FromTo(i1, i2) ->
    let rec loop i =
      if i <= i2 then
        cont (Int i) (fun () -> loop (i+1))
      else
        econt ()
    in loop i1
  | Write e ->
    eval e (fun v ->
      fun econt1 -> (write v; cont v econt1))
      econt
  | If(e1, e2, e3) ->
    eval e1 (fun _ -> fun _ ->
      eval e2 cont econt)
      (fun () -> eval e3 cont econt)
  | ...

If success, throw out e1 and evaluate e2
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | ...
    | FromTo(i1, i2) ->
      let rec loop i =
        if i <= i2 then
          cont (Int i) (fun () -> loop (i+1))
        else
          econt ()
      in loop i1
    | Write e ->
      eval e (fun v ->
        eval e (fun _ ->
          fun _ -> eval e2 cont econt)
        (fun () -> eval e3 cont econt))
    | ...

If failure, evaluate e3
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | ...
let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | ... |
  | And(e1, e2) ->
    eval e1 (fun _ -> fun econt1 -> eval e2 cont econt)
    eval e2 (fun () -> eval e2 cont econt) econt
let rec eval (e : expr) (cont : cont) (econt : econt) =
match e with
| ... |
| And(e1, e2) ->
  eval e1 (fun _ -> fun econt1 -> eval e2 cont econt1) econt
| Or(e1, e2) ->
  eval e1 cont (fun () -> eval e2 cont econt)

Represents e1 | e2: do e2 after e1 fails (each left then each right)
let rec eval (e : expr) (cont : cont) (econt : econt) =
  match e with
  | ...,
  | And(e1, e2) ->
    eval e1 (fun _ -> fun econt1 -> eval e2 cont econt1) econt
  | Or(e1, e2) ->
    eval e1 cont (fun () -> eval e2 cont econt)
  | Seq(e1, e2) ->
    eval e1 (fun _ -> fun econt1 -> eval e2 cont econt)
    (fun () -> eval e2 cont econt)

Represents e1 ; e2: do e2 no matter what, no backtracking on left
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | ... |
    | And(e1, e2) ->
      eval e1 (fun _ -> fun econt1 -> eval e2 cont econt) econt
    | Or(e1, e2) ->
      eval e1 cont (fun () -> eval e2 cont econt)
    | Seq(e1, e2) ->
      eval e1 (fun _ -> fun econt1 -> eval e2 cont econt)
      (fun () -> eval e2 cont econt)
    | Every e ->
      eval e (fun _ -> fun econt1 -> econt1 ())
      (fun () -> cont (Int 0) econt)

Take result, ignore it, ask for one more
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | ... |
    | And(e1, e2) ->
        eval e1 (fun _ -> fun econt1 -> eval e2 cont econt1) econt
    | Or(e1, e2) ->
        eval e1 cont (fun () -> eval e2 cont econt)
    | Seq(e1, e2) ->
        eval e1 (fun _ -> fun econt1 -> eval e2 cont econt)
            (fun () -> eval e2 cont econt)
    | Every e ->
        eval e (fun _ -> fun econt1 -> econt1 ()
            (fun () -> cont (Int 0) econt))

Finally succeed with 0
THIS WEEK’S LECTURE

▶ PLCSD chapter 11
▶ Exercises 11.1, 11.2, 11.3, 11.4, 11.8

NEXT WEEK

▶ PLCSD chapter 12