Continuations: Exceptions, backtracking, Micro-Icon

David Christiansen[1]

Monday, 29 October, 2012

[1] based on slides by Peter Sestoft

IT University of Copenhagen

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

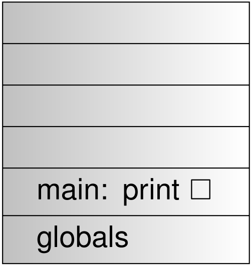
- Throwing exceptions
- Handling exceptions

MICRO-ICON

- Micro-Icon introduction
- Micro-Icon interpreter

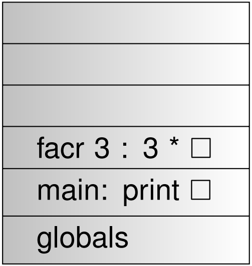
Stack frames and continuations

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

    facr 3
```



Stack frames and continuations

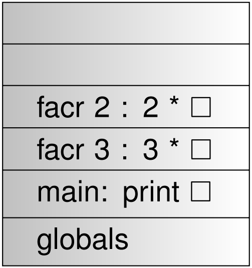
```
let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

    facr 3
⟹  3 * facr (3 - 1)
```



Stack frames and continuations

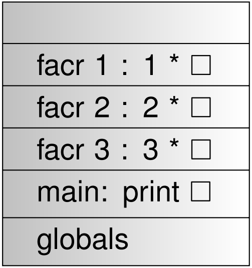
```
let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)
```

```
    facr 3
⟹  3 * facr (3 - 1)
⟹  3 * (2 * facr (2 - 1))
```



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)))
```



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))
```

facr 0 : 1
facr 1 : 1 * □
facr 2 : 2 * □
facr 3 : 3 * □
main: print □
globals

Stack frames and continuations

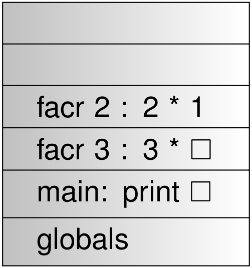
```
let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)

    facr 3
⟹  3 * facr (3 - 1)
⟹  3 * (2 * facr (2 - 1))
⟹  3 * (2 * (1 * facr (1 - 1)))
⟹  3 * (2 * (1 * 1))
⟹  3 * (2 * 1)
```

facr 1 : 1*1
facr 2 : 2 * □
facr 3 : 3 * □
main: print □
globals

Stack frames and continuations

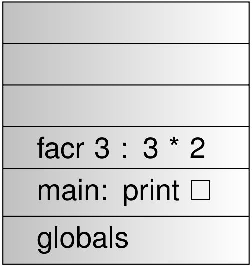
```
let rec facr n =
  if n = 0
  then 1
  else n * facr (n - 1)
```

```
    facr 3
⟹  3 * facr (3 - 1)
⟹  3 * (2 * facr (2 - 1))
⟹  3 * (2 * (1 * facr (1 - 1)))
⟹  3 * (2 * (1 * 1))
⟹  3 * (2 * 1)
⟹  3 * 2
```



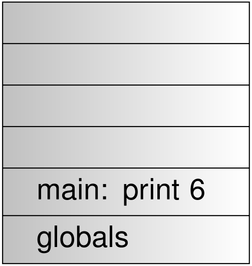
Stack frames and continuations

```
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 * 1)
⟹  3 * 2
⟹  6
```



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 * 1)
⟹  3 * 2
⟹  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

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))
```

Deriving a CPS facr

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

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 ???
  else ???
```

- Add continuation argument

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) <n * □>
```

- 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)
```

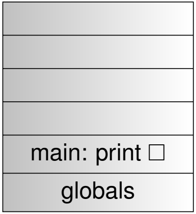
```
let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))
```

- `facc n k = k (facr n)`
- `facr n = facc n (fun v -> v)`

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

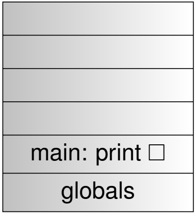
Evaluating facc

```
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
```



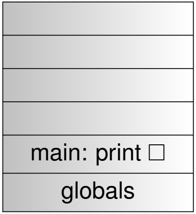
Evaluating facc

```
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))
```



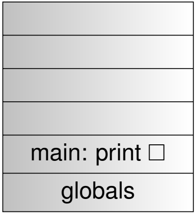
Evaluating facc

```
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))
```

Evaluating facc

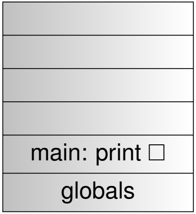



```
let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

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))
```

Evaluating facc

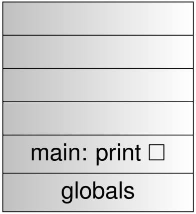
```
let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

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
```

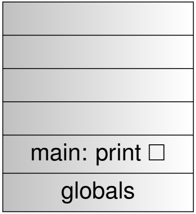
Evaluating facc

```
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)
```

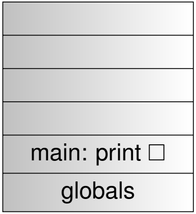
Evaluating facc




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

let id x = x
```

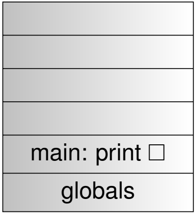
```
    facc 3 id
⟹  facc 2 (fun v -> id (3 * v))
⟹  facc 1 (fun w -> (fun v -> id (3 * v)) (2 * w))
⟹  facc 0 (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u))
⟹  (fun u -> (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * u)) 1
⟹  (fun w -> (fun v -> id (3 * v)) (2 * w)) (1 * 1)
⟹  (fun w ->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

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)
```

Evaluating facc

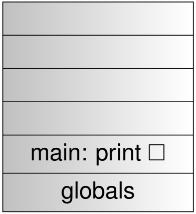



```
let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

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
```

Evaluating facc

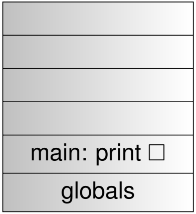
```
let rec facc n k =
  if n = 0
  then k 1
  else facc (n - 1) (fun v -> k (n * v))

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)
```

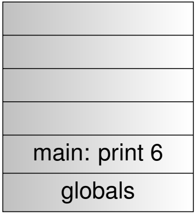
Evaluating facc

```
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
```

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
```

Exercise

Convert the following function to CPS.

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

Hint: start with

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

Break

Overview

Tail recursion and iteration

Rewrite `facr` with accumulator:

```
let rec faci n r =
  if n = 0
  then r
  else faci (n - 1) (r * n)

    faci 3 1
```

```
faci n r = r * facr n
facr n = faci n 1
```

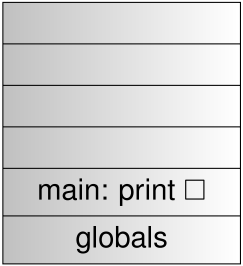
Tail recursion and iteration

Rewrite `facr` with accumulator:

```
let rec faci n r =
  if n = 0
  then r
  else faci (n - 1) (r * n)
```

```
faci n r = r * facr n
facr n = faci n 1
```

```
    faci 3 1
⟹  faci 2 3
```



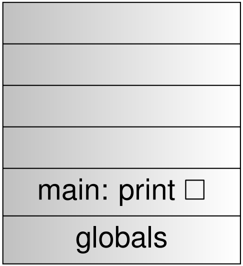
Tail recursion and iteration

Rewrite `facr` with accumulator:

```
let rec faci n r =
  if n = 0
  then r
  else faci (n - 1) (r * n)
```

```
faci n r = r * facr n
facr n = faci n 1
```

```
   faci 3 1
⟹ faci 2 3
⟹ faci 1 6
```



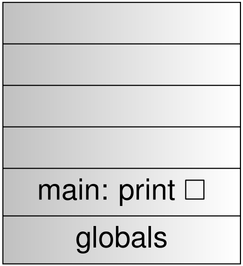
Tail recursion and iteration

Rewrite `facr` with accumulator:

```
let rec faci n r =
  if n = 0
  then r
  else faci (n - 1) (r * n)
```

```
faci n r = r * facr n
facr n = faci n 1
```

```
   faci 3 1
⟹ faci 2 3
⟹ faci 1 6
⟹ faci 0 6
```



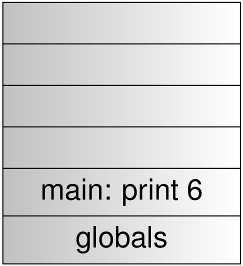
Tail recursion and iteration

Rewrite `facr` with accumulator:

```
let rec faci n r =
  if n = 0
  then r
  else faci (n - 1) (r * n)
```

```
faci n r = r * facr n
facr n = faci n 1
```

```
   faci 3 1
⟹ faci 2 3
⟹ faci 1 6
⟹ faci 0 6
⟹ 6
```



Continuations vs accumulating parameters

- Which of `facr n`, `facc n k`, `faci n r` are tail-recursive?

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`?

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:

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`

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)
    ```

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)
```

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)
```

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
```

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`

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

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

CPS in Java

CONTINUATIONS

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

CPS in Java

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

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

Why CPS?

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

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

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

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

Interpreter for throwing exceptions (part 1)

```
let rec coEval1 e env (cont : int -> answer) : answer =
    match e with
    | CstI i -> cont i
```

Interpreter for throwing exceptions (part 1)

```
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"
```

Interpreter for throwing exceptions (part 1)

```
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)
              | ... ))
```

Interpreter for throwing exceptions (part 1)

```
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
```

Interpreter for throwing exceptions (part 2)

```
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)
    | ...
```

Interpreter for throwing exceptions (part 2)

```
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:
  `econt :  exn -> answer`

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

Interpreter for handling exceptions

- Add an error continuation to the interpreter:
  `econt :  exn -> answer`

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

- The error continuation decides whether or not to handle the exception

Interpreter for throwing and handling exceptions

Non-exception evaluation is as before:

```
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
    | ...
```

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
```

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

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
```

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

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 error doesn't match, pass it up to next error handler

Break

Overview

Expressions giving multiple results; the Icon language

Expression	Results	Output	Comment
`5`	`5`		Constant
`write 5`	`5`	`5`	Constant, side effect
`(1 to 3)`	`1 2 3`		Range, 3 results
`write (1 to 3)`	`1 2 3`	`1`	Side effect
`every (write (1 to 3))`	`0`	`1 2 3`	Force all results
`(1 to 0)`			Empty range, no res.
`&fail`			No results
`(1 to 3)+(4 to 6)`	`5 6 7 6 7 8 7 8 9`		All combinations

Expressions giving multiple results; the Icon language

Expression	Results	Output	Comment
`3 < 4`	`4`		Comparison succeeds
`4 < 3`			Comparison fails
`3 < (1 to 5)`	`4 5`		Succeeds twice
`(1 to 3) | (4 to 6)`	`1 2 3 4 5 6`		Each left, each right
`(1 to 3) & (4 to 6)`	`4 5 6 4 5 6 4 5 6`		Each right for each left
`(1 to 3) ; (4 to 6)`	`4 5 6`		No backtracking to left

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

Micro-Icon interpreter

The interpreter takes two continuations:

FAILURE CONTINUATION :

```
econt :  unit -> answer
```

called when there are no (more) results

SUCCESS CONTINUATION :

```
cont :  value -> econt -> answer
```

called when there is one (more) result

Micro-Icon interpreter

The interpreter takes two continuations:

FAILURE CONTINUATION :

```
econt :  unit -> answer
```

called when there are no (more) results

SUCCESS CONTINUATION :

```
cont :  value -> econt -> answer
```

called when there is one (more) result

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

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

Micro-Icon Interpreter

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
```

Micro-Icon Interpreter

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | CstI i -> cont (Int i) econt
    | CstS s -> cont (Str s) econt
```

Succeed with a constant

Micro-Icon Interpreter

```
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
    | ...
```

Continuation for left argument e1

Micro-Icon Interpreter

```
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
    | ...
```

Continuation for right argument e2

Micro-Icon Interpreter

```
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

Micro-Icon Interpreter

```
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
    | ...
```

Call provided error if not less than

Micro-Icon Interpreter

```
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

Micro-Icon Interpreter

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | ...
```

Micro-Icon Interpreter

```
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
```

Handle `1 to 3`

Micro-Icon Interpreter

```
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
```

While values are left, send them to the success continuation

Micro-Icon Interpreter

```
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
```

`cont` gets the next loop iteration in case of failure

Micro-Icon Interpreter

```
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
```

When done looping, go back to previous failure continuation

Micro-Icon Interpreter

```
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

Micro-Icon Interpreter

```
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

Micro-Icon Interpreter

```
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 failure, evaluate e3

Micro-Icon interpreter

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | ...
```

Micro-Icon interpreter

```
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
```

Represents `e1 & e2`: combine each `e1` with each `e2`

Micro-Icon interpreter

```
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)

Micro-Icon interpreter

```
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

Micro-Icon interpreter

```
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)
```

Take result, ignore it, ask for one more

Micro-Icon interpreter

```
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

Reading and homework

This week's lecture

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

Next week

- PLCSD chapter 12