Translation / SLang

```
(Lecture7) Overview
simple, denotational functions
    [v0: _eval_ that takes functions as argument]

(Lecture8)
→ cps (higher-order functions)
    [ v1: eval with continuation k ]
→ defun (mutual recursive functions)
    [ v2: _eval + apply_ ]
→ tag list (stack) continuation
    [ v3: _tag type = SUB2, SUB, ADD_ ]
===Interpreter 0 ✔===

→ split stacks  ( tail recursive functions)
    [ v4: via state datatype,
        eval =
            | (EVAL, n, k → ...)
            | (APP, n, (SUB2 m)::k→ ...)
    ]
→ small step stack machine (sequential first order logic)
    [ v5: _step + driver_ ]

(Lecture9)
→ two stacks (data values and code expressions)
    [ v6: _directive_stack * value_stack_ ]
===Interpreter 1 ✔===

→ two stages (compilation and evaluation / interpretation)
    [ v7: _code * value_stack_, where _type code = instr list_ ]
===Interpreter 2 ✔===

(Lecture10)
→ linear code (global instruction array)
    [ cp, Label L, GOTO L ]
===Interpreter 3 ✔===

(Lecture11)
→ Addressable stack, heap
    [closures = code pointer + free variables]
=== Jargon VM ===

(Lecture12) Garbage collection
(Lecture13) Opt
(Lecture14) Exception
(Lecture15) Linking
(Lecture16) Bootstrapping
```

Extra Notes on CPS

```
CPS Invariant,
eval_cps e c = c (eval e)

Now consider,
f (g x) ⇝ g x (fun y → f y k)

Breakdown
eval f (g x) ⇝ eval_cps g x (fun y → eval_cps f y k)
             ⇝ eval_cps g x k0              [ first continuation k0 = (fun y → eval_cps f y k) ]
             ⇝ k0 (eval g x)                [ by Invariant ]
             ⇝ (fun y → eval_cps f y k) (eval g x)
                                            [ intermediate res y = g(x) ]
             ⇝ eval_cps f y k               [ second continuation k = ID ]
             ⇝ k f y                        [ by Invariant ]
             ⇝ f y
             ⇝ f (g x)
```

Property

- explicit evaluation order
- every call is a tail call
    - provided that function as values / first-class functions

✓ Q y17p23q4

Y2017P3Q4. CPS & Defunctionalisation

ADD_PAIR : PAIR → value.

(* apply a named function *) ADD_INT : INT → value

```
type expr  = | Integer of int | Pair of expr * expr | Apply of string * expr
type value = | INT of int     | PAIR of value * value
```

```
let rec eval = function
 | Integer n → INT n
 | Pair (e1, e2) → PAIR (eval e1, eval e2)
 | Apply (f, e) → eval-function (f, eval e)
```

let V = eval e in
C (eval-function (f, V))

1. Add a continuation parameter c to each function, return value.

(a)

```
let rec eval-cps C = function
 | Integer n → C (INT n)
 | Pair (e1, e2) → eval-cps (fun V1 → (* PAIR 1 *) λV1, e2, C
                    eval-cps (fun V2 → (* PAIR 2 *) λV1, V2, C
                       C (PAIR (V1, V2))) e2) e1
 | Apply (f, e) → eval-cps (fun V → C (eval-function (f, V))) e ...... apply-cnt
                            (* FUNC *) -- λf, c.
```

let V1 = eval e1 in
V2 = eval e2 in
C (PAIR (V1, V2))

Thus

```
let eval_2 e = eval-cps (fun x → x) e .
```

(* ID *) .... part b.

eval : expr → value.

eval-cps : (fun) → expr → value

$\boxed{\text{(IH)} \quad \underline{C} \text{ (eval } e\text{)} = \text{eval\_cps } C \; e}$

(b) Eliminate high-order continuations.

1. Add a constructor to cnt for each fun (* Cnt *) λ... (free variables)

```
type cnt =
 | ID
 | PAIR1 of expr * cnt
 | PAIR2 of value * cnt
 | FUNC of string * cnt
```

Call apply-cnt at every application of continuation.

```
let rec eval-cps-dfn c = function
 | Interger n → apply-cnt C (INT n)                          ×0
 | Pair (e1, e2) → eval-cps-dfn (PAIR1 (e2, C)) e1           ..
 | Apply (f, e) → eval-cps-dfn (FUNC (f, c)) e               ×1

and apply-cnt = function
 | (ID, V) → V
 | (PAIR1 (e2, C), V1) → eval-cps-dfn (PAIR2 (V1, C)) e2     ×2
 | (PAIR2 (V1, C), V2) → apply-cnt (C, PAIR (V1, V2))
 | (Func (f, c), V) → apply-cnt (c, eval-function (f, V))

let eval_3 e = eval-cps-dfn ID e .
```

Mutually recursive.

fun V1 ↦ eval-cps ... e2
fun V2 ↦ C (PAIR (V1, V2))
fun V ↦ C (eval-function (f, V))

for all.

eval-cps-dfn : cnt → expr → value
apply-cnt : cnt * value → value
eval_3 : expr → value