CSE216 Programming Abstractions

Simple Programming Language (SPL)

YoungMin Kwon

Simple Programming Language (SPL)

- Metalinguistic Abstraction

> … It's in words that the magic is–Abracadabra, Open Sesame and the rest–but the magic words in one story aren't magical in the next. The real magic is to understand which words work, and when, and for what; the trick is to learn the trick.
> … And those words are made from the letters of our alphabet: a couple-dozen squiggles we can draw with the pen. This is the key! And the treasure, too, if we can only get our hands on it! It's as if—as if the key to the treasure *is* the treasure!
>
> John Barth, *Chimera*

What is a Programming Language

- To describe a language, we need to understand
  - Its syntax and semantics
- Syntax
  - The *structure* of expressions
  - E.g. 1 * 2 + 3 - 4
- Semantics
  - The *meaning* of such expressions
    - 2 ^ 3: 2 xor 3 or 2^3 ?
  - How such expressions are evaluated

Syntax of a Language

- We use grammar to describe the syntax of a language
- Grammar: a set of recursive rules that define what *forms* or *shapes* are permitted
- Grammars are often described in *Backus-Naur Form* (BNF)
  - BNF is a metalanguage: a language to describe a language

Syntax of a Language

- A simple BNF for arithmetic expressions

```
N ::= … |-2 |-1 | 0 | 1 | 2 | …
⊕ ::= + | - | * | /
E ::= N | E1 ⊕ E2
```

  - E.g
    - 1 is a valid expression
    - 2 + 3 - 4 is a valid expression
    - 2 + + 3 is not a valid expression

Semantics of a Language

- Semantics of a language
  - A process for repeatedly reducing an expression
  - Reducible expression: an expression that can be further evaluated

Semantics of a Language

- Logical rules to derive meanings from expressions
  - $\rightarrow \subset E \times E$: *reduction* relation
  - We write $e \rightarrow e'$ for $(e, e') \in \rightarrow$
- Inference rules

$$e \rightarrow e' \Rightarrow e \oplus e'' \rightarrow e' \oplus e''$$
$$e \rightarrow e' \Rightarrow e'' \oplus e \rightarrow e'' \oplus e'$$

$$\frac{e \rightarrow e' \qquad e' \rightarrow e''}{e \rightarrow e''} \qquad \frac{n, n', n'' \in N \qquad n = n' \oplus n''}{n' \oplus n'' \rightarrow n}$$

Semantics of a Language

- Example
  - 1 + 2 * 3 - 4 (without operator precedence)

$$\frac{1 + 2 = 3}{\dfrac{1 + 2 \rightarrow 3}{1 + 2 * 3 - 4 \rightarrow 3 * 3 - 4}}$$

$$\frac{3 * 3 = 9}{\dfrac{3 * 3 \rightarrow 9}{3 * 3 - 4 \rightarrow 9 - 4}}$$

$$\frac{9 - 4 = 5}{9 - 4 \rightarrow 5}$$

$$e \rightarrow e' \Rightarrow e \oplus e'' \rightarrow e' \oplus e''$$

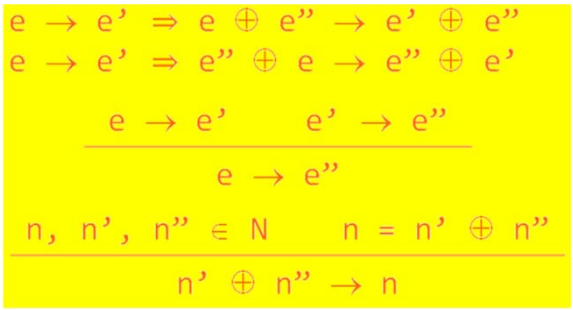
$$e \rightarrow e' \Rightarrow e'' \oplus e \rightarrow e'' \oplus e'$$

$$\frac{e \rightarrow e' \qquad e' \rightarrow e''}{e \rightarrow e''}$$

$$\frac{n, n', n'' \in N \qquad n = n' \oplus n''}{n' \oplus n'' \rightarrow n}$$

Semantics of a Language

- Example
  - By the transitivity



$$\frac{3 * 3 - 4 \rightarrow 9 - 4 \qquad 9 - 4 \rightarrow 5}{3 * 3 - 4 \rightarrow 5}$$

$$\frac{1 + 2 * 3 - 4 \rightarrow 3 * 3 - 4 \qquad 3 * 3 - 4 \rightarrow 5}{1 + 2 * 3 - 4 \rightarrow 5}$$

Semantics of a Language

- Program as a proof
  - E.g. prove that

$$1 + 2 * 3 - 4 \rightarrow 5$$

$$e \rightarrow e' \Rightarrow e \oplus e'' \rightarrow e' \oplus e''$$

$$e \rightarrow e' \Rightarrow e'' \oplus e \rightarrow e'' \oplus e'$$

$$\frac{e \rightarrow e' \qquad e' \rightarrow e''}{e \rightarrow e''}$$

$$\frac{n, n', n'' \in N \qquad n = n' \oplus n''}{n' \oplus n'' \rightarrow n}$$

$$\begin{array}{rl}
(1 + 2, 3) & \in \rightarrow \\
(1 + 2 * 3 - 4, 3 * 3 - 4) & \in \rightarrow \\
(3 * 3, 9) & \in \rightarrow \\
(1 + 2 * 3 - 4, 9 - 4) & \in \rightarrow \\
(9 - 4, 5) & \in \rightarrow \\
(1 + 2 * 3 - 4, 5) & \in \rightarrow
\end{array}$$

$$\begin{array}{rl}
1 + 2 & \rightarrow 3 \\
1 + 2 * 3 - 4 & \rightarrow 3 * 3 - 4 \\
3 * 3 & \rightarrow 9 \\
1 + 2 * 3 - 4 & \rightarrow 9 - 4 \\
9 - 4 & \rightarrow 5 \\
1 + 2 * 3 - 4 & \rightarrow 5
\end{array}$$

Simple Programming Language (SPL)

- Designing a programming language
  - To better understand programming abstractions
  - To better understand to the tool we are using: programming languages
  - Metalinguistic abstraction: establishing a new language → a powerful strategy for controlling complexity

SPL: Syntax

- Syntax of SPL

```
type expr = B of bool   (*boolean*)
          | N of int    (*number*)
          | V of string (*variable*)
          (*arithmetic exprs*)
          | Add of expr * expr | Sub of expr * expr
          (*predicates*)
          | Equ of expr * expr | Leq of expr * expr
          (*logical exprs*)
          | And of expr * expr | Or  of expr * expr | Not of expr
          (*conditional expr*)
          | If  of expr * expr * expr
          (*function definition: parameter, body*)
          | Fun of string * expr
          (*closure: parameter, environment, body*)
          | Clo of string * (string * expr) list * expr
          (*function application: operator, operand*)
          | App of expr * expr;;
```

SPL: Semantics

- Constants: numbers and booleans

$$\langle \texttt{N n}, \sigma \rangle \rightarrow \langle \texttt{N n}, \sigma \rangle$$
$$\langle \texttt{B b}, \sigma \rangle \rightarrow \langle \texttt{B b}, \sigma \rangle$$

- Variables

$$\langle \texttt{V v}, \sigma \rangle \rightarrow \langle \sigma(\texttt{v}), \sigma \rangle$$

  - σ is a state
    - It is a function from a string (name) to a value
    - Value: number, boolean, closure

SPL: Semantics

- Binary operations

$$\frac{\langle e,\ \sigma\rangle \rightarrow \langle N\ a,\ \sigma\rangle \qquad \langle e',\ \sigma\rangle \rightarrow \langle N\ b,\ \sigma\rangle}{\langle Add\ (e,\ e'),\ \sigma\rangle \rightarrow \langle N\ a + b,\ \sigma\rangle}$$

$$\frac{\langle e,\ \sigma\rangle \rightarrow \langle N\ a,\ \sigma\rangle \qquad \langle e',\ \sigma\rangle \rightarrow \langle N\ b,\ \sigma\rangle}{\langle Sub\ (e,\ e'),\ \sigma\rangle \rightarrow \langle N\ a - b,\ \sigma\rangle}$$

$$\frac{\langle e,\ \sigma\rangle \rightarrow \langle N\ a,\ \sigma\rangle \qquad \langle e',\ \sigma\rangle \rightarrow \langle N\ b,\ \sigma\rangle}{\langle Equ\ (e,\ e'),\ \sigma\rangle \rightarrow \langle B\ a = b,\ \sigma\rangle}$$

$$\frac{\langle e,\ \sigma\rangle \rightarrow \langle N\ a,\ \sigma\rangle \qquad \langle e',\ \sigma\rangle \rightarrow \langle N\ b,\ \sigma\rangle}{\langle Leq\ (e,\ e'),\ \sigma\rangle \rightarrow \langle B\ a \leq b,\ \sigma\rangle}$$

SPL: Semantics

- Binary operations

$$\frac{\langle e,\ \sigma\rangle \rightarrow \langle B\ a,\ \sigma\rangle \qquad \langle e',\ \sigma\rangle \rightarrow \langle B\ b,\ \sigma\rangle}{\langle And\ (e,\ e'),\ \sigma\rangle \rightarrow \langle B\ a \wedge b,\ \sigma\rangle}$$

$$\frac{\langle e,\ \sigma\rangle \rightarrow \langle B\ a,\ \sigma\rangle \qquad \langle e',\ \sigma\rangle \rightarrow \langle B\ b,\ \sigma\rangle}{\langle Or\ (e,\ e'),\ \sigma\rangle \rightarrow \langle B\ a \vee b,\ \sigma\rangle}$$

$$\langle e,\ \sigma\rangle \rightarrow \langle B\ a,\ \sigma\rangle \Rightarrow \langle Not\ e,\ \sigma\rangle \rightarrow \langle B\ \neg a,\ \sigma\rangle$$

- Short circuit evaluation of And and Or
  - In many programming languages like Java, C, ml, …
  - And(e, e'): if is e false, e' is not evaluated
  - Or(e, e'): if e is true, e' is not evaluated

SPL: Semantics

- Conditional operation

$$\frac{\langle e,\ \sigma\rangle \rightarrow \langle B\ a,\ \sigma\rangle}{\langle \texttt{If}\ (e,\ e',\ e''),\ \sigma\rangle \rightarrow \langle \texttt{If}\ (B\ a,\ e',\ e''),\ \sigma\rangle}$$

$$\langle \texttt{If}\ (B\ \texttt{true},\ e',\ e''),\ \sigma\rangle \rightarrow \langle e',\ \sigma\rangle$$

$$\langle \texttt{If}\ (B\ \texttt{false},\ e',\ e''),\ \sigma\rangle \rightarrow \langle e'',\ \sigma\rangle$$

SPL: Semantics

- Function definition and Closure

$$\langle \texttt{Fun (v, e)}, \sigma \rangle \rightarrow \langle \texttt{Clo (v, } \sigma \texttt{, e)}, \sigma \rangle$$

  - A closure has the state where the function is defined

- Function application

$$\frac{\langle e, \sigma \rangle \rightarrow \langle a, \sigma \rangle \qquad \langle e', \sigma'[v\backslash a] \rangle \rightarrow \langle b, \sigma'[v\backslash a] \rangle}{\langle \texttt{App (Clo (v, } \sigma' \texttt{, e'), e)}, \sigma \rangle \rightarrow \langle b, \sigma \rangle}$$

$$\text{where } \sigma[v\backslash a](x) = \begin{cases} a & \text{if } x = v \\ \sigma(x) & \text{if } x \neq v \end{cases}$$

SPL: Evaluation

- Evaluation of SPL (without variables)

```
let rec eval expr env =
    let getNum  = function N n -> n | assert false in
    let getBool = function B b -> b | assert false in

    match expr with
    | B b -> B b
    | N n -> N n

    | Add (a, b)   -> eval a env |> fun x ->
                      eval b env |> fun y ->
                      N (getNum x + getNum y)
    | Sub (a, b)   -> eval a env |> fun x ->
                      eval b env |> fun y ->
                      N (getNum x - getNum y)
```

SPL: Evaluation

```
| Equ (a, b)    -> eval a env |> fun x ->
                   eval b env |> fun y ->
                   B (getNum x = getNum y)
| Leq (a, b)    -> eval a env |> fun x ->
                   eval b env |> fun y ->
                   B (getNum x <= getNum y)

| And (a, b)    -> eval a env |> fun x ->
                   eval b env |> fun y ->
                   B (getBool x && getBool y)
| Or (a, b)     -> eval a env |> fun x ->
                   eval b env |> fun y ->
                   B (getBool x || getBool y)
| Not a         -> eval a env |> fun x ->
                   B (not (getBool x))

| If (c, t, f) -> eval c env |> fun x ->
                   if getBool x then eval t env
                                else eval f env
| _ -> assert false
```

SPL: Evaluation

- Testing

```
# eval (Leq (Add (N 2, N 1), Sub (N 2, N 1))) [];;
- : expr = B false

# eval (Add (Add (N 2, N 1), Sub (N 2, N 1))) [];;
- : expr = N 4

# eval (If (B true, N 2, N 3)) [];;
- : expr = N 2

# eval (If (Leq (N 1, N 2), Add (N 1, N 2), Sub (N 1, N 2))) [];;
- : expr = N 3
```

SPL: Evaluation

- Evaluation with variables
  - Static scoping rule
  - Environment model
- Evaluating function definition
  - $\rightarrow$ Closure
    - Formal parameter name
    - Function body
    - Environment where the function is defined

Environment Model of Evaluation

- Environment
  - A sequence of frames
  - Each frame is a table of bindings
  - Each binding associates a name with its value

- Value of a variable w.r.t. an environment
  - Each frame has a pointer to its enclosing environment
  - Find the first frame that has a binding for the variable in the environment
  - The value of the variable is the value of the binding

Environment Model of Evaluation

- Value of a variable

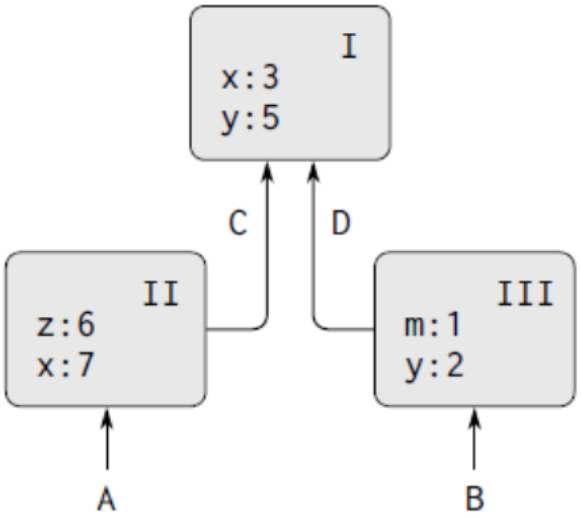
```
# let f x y =

    let g x z =
        x + y + z in

    let h m y =
        m + x + y in

    x + y + g 6 7 + h 1 2

# f 3 5;;
- : int = 32
```



Rules for Evaluation

- To evaluate a combination
  - Evaluate the subexpressions of the combination
  - Apply the value of the operator subexpression to the values of the operand subexpressions

- Environment model vs substitution model

Environment model

```
let square x = x * x
square (2 + 3)
⇒ square 5
⇒ 5 * 5
⇒ 25
```

Substitution model

```
let square x = x * x
square (2 + 3)
⇒ (2 + 3) * (2 + 3)
⇒ 5 * 5
⇒ 25
```

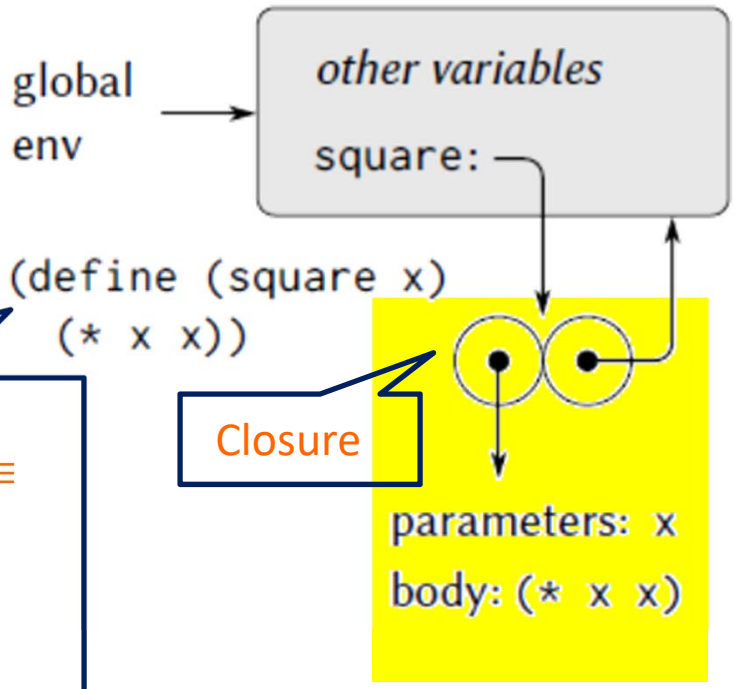
Rules for Evaluation

- Procedure application to arguments
  - Construct a frame
  - Bind the formal parameters to actual parameters
  - Evaluate the body
- Procedure creation
  - Evaluate lambda (fun) expression in an environment
  - Resulting procedure is a closure (a pair) of
    - Text of the lambda expression
    - Pointer to the environment the procedure is created

Rules for Evaluation

- E.g.) defining a procedure in an environment

```
let square =
    fun x -> x * x;;
```



Scheme syntax:

```
(define (square x) (* x x)) ≡
(define square
    (lambda (x) (* x x))
```

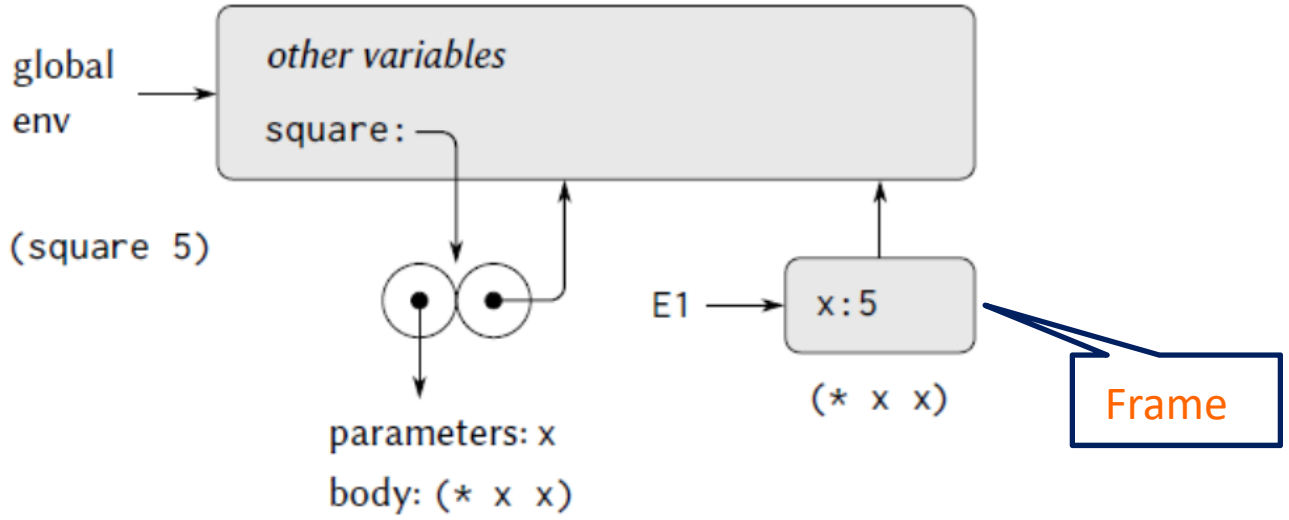
Equivalent expression in ml

```
let square x = x * x ≡
let square = fun x -> x * x
```

Rules for Evaluation

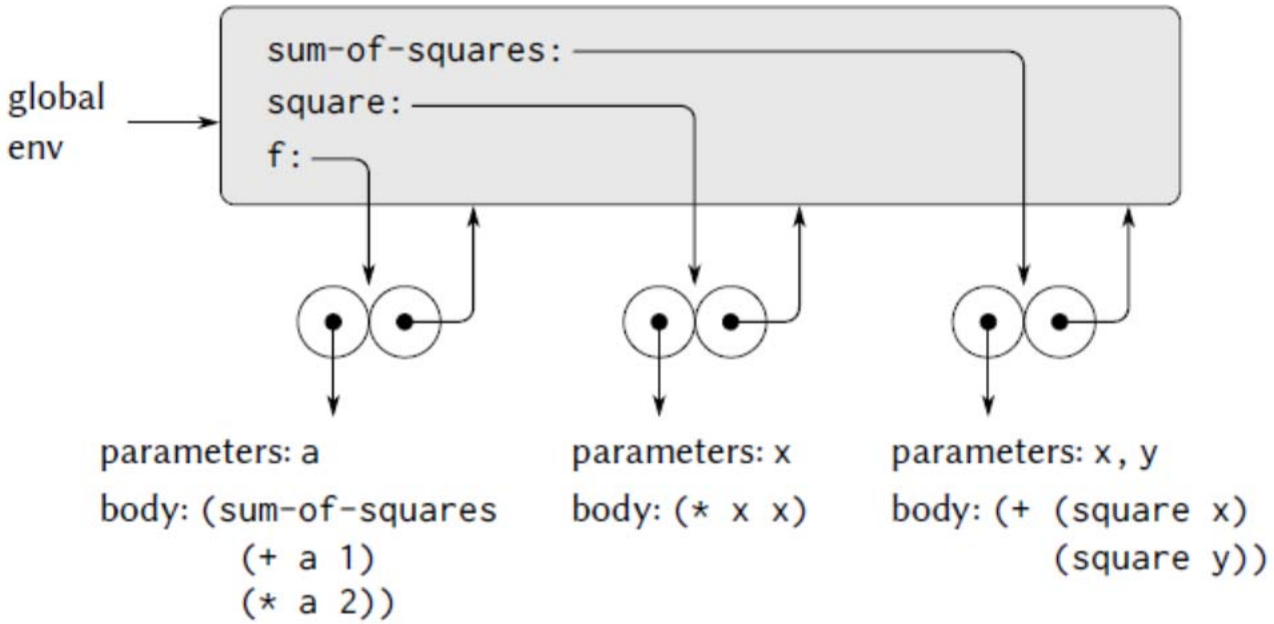
- E.g.) Applying the procedure
  - A frame is added to the environment

```
square 5;;
```

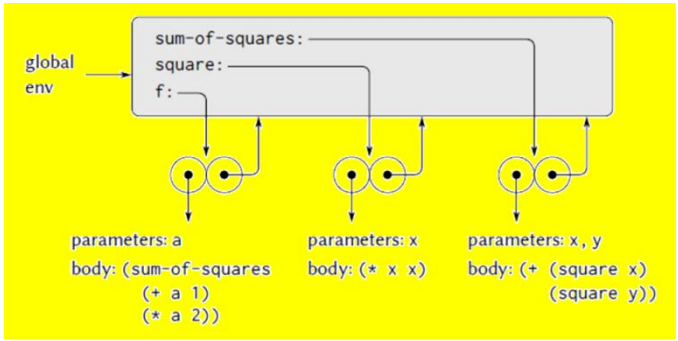


Applying Simple Procedures

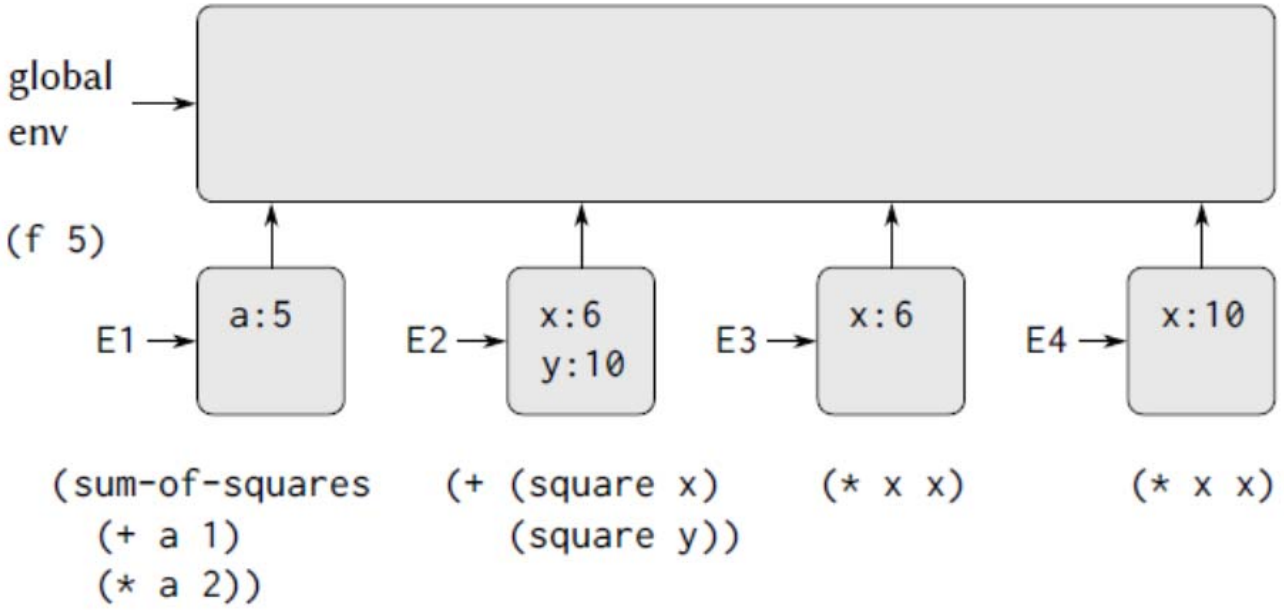
```
let square
    = fun x -> x * x;;
let sum_of_squares
    = fun x y -> square x + square y;;
let f
    = fun a -> sum_of_squares (a+1) (a*2);;
```



Applying Simple Procedures




```
f 5;;
```



Frames as the Repository of Local State

- `make_withdraw` example

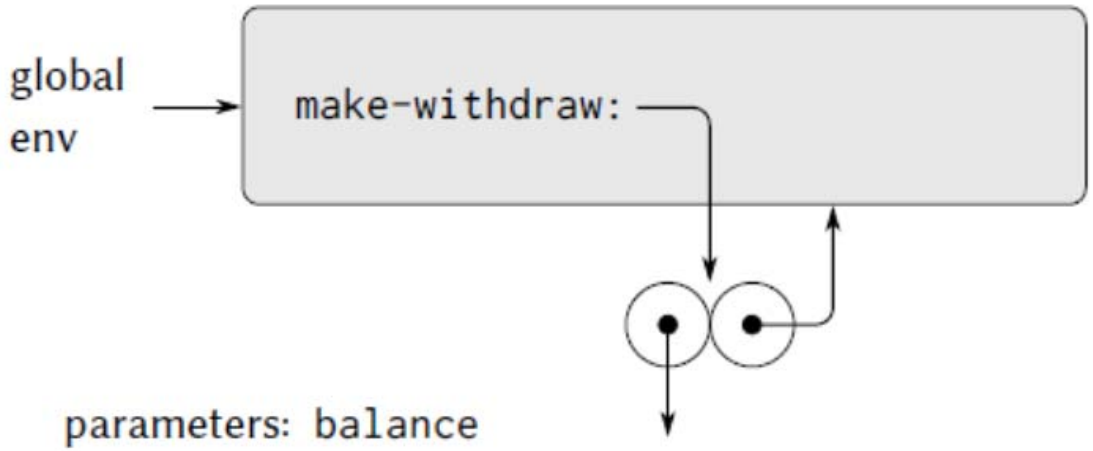
```
let make_withdraw balance =
    fun amount ->
        if !balance >= amount
        then begin
            balance := !balance - amount;
            !balance
        end
        else assert false;;

let w1 = make_withdraw (ref 100) ;;
w1 50;;

let w2 = make_withdraw (ref 100) ;;
w2 50;;
```

Frames as the Repository of Local State

- Result of defining `make_withdraw` in the global environment

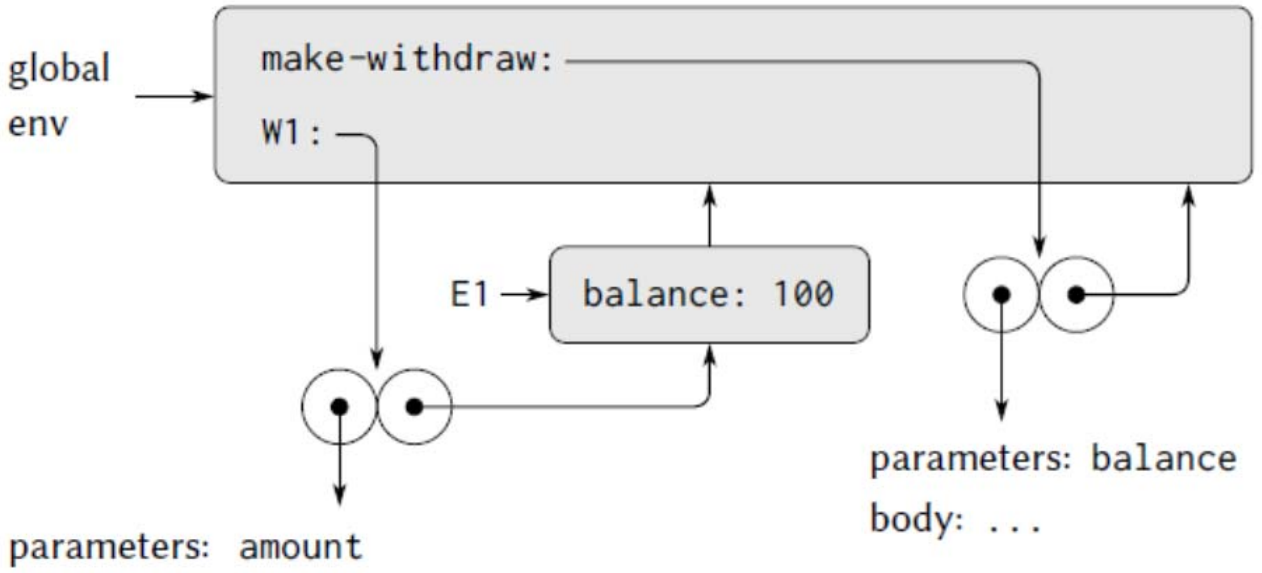


parameters: balance

```
body: fun amount ->
          if !balance >= amount
          then begin
              balance := !balance - amount;
              !balance
          end
          else assert false;;
```

Frames as the Repository of Local State

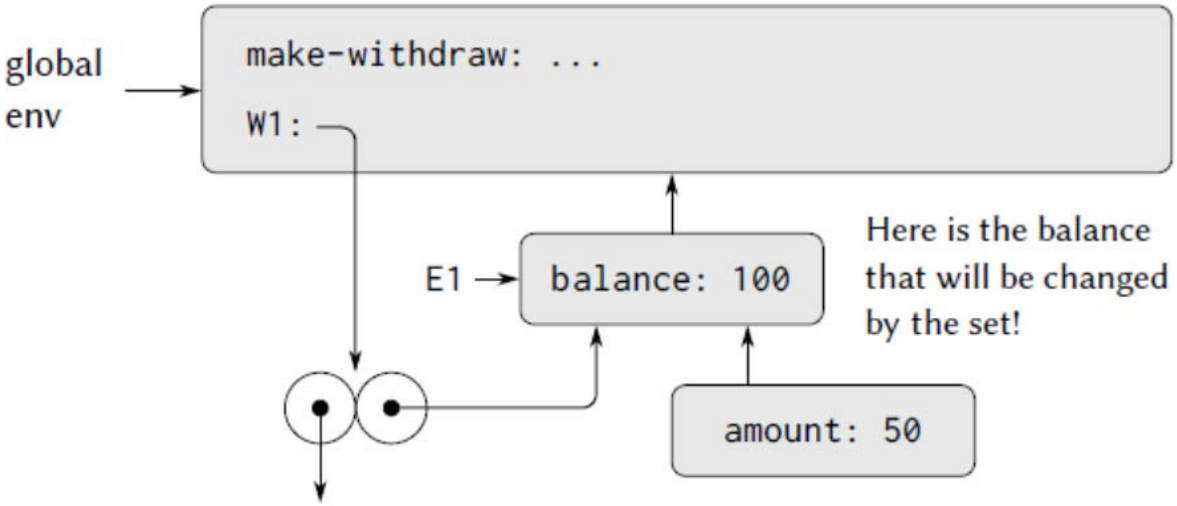
- Result of evaluating `let w1 = make_withdraw (ref 100)`




```
if !balance >= amount
then begin
    balance := !balance - amount;
    !balance
end
else assert false;;
```

Frames as the Repository of Local State

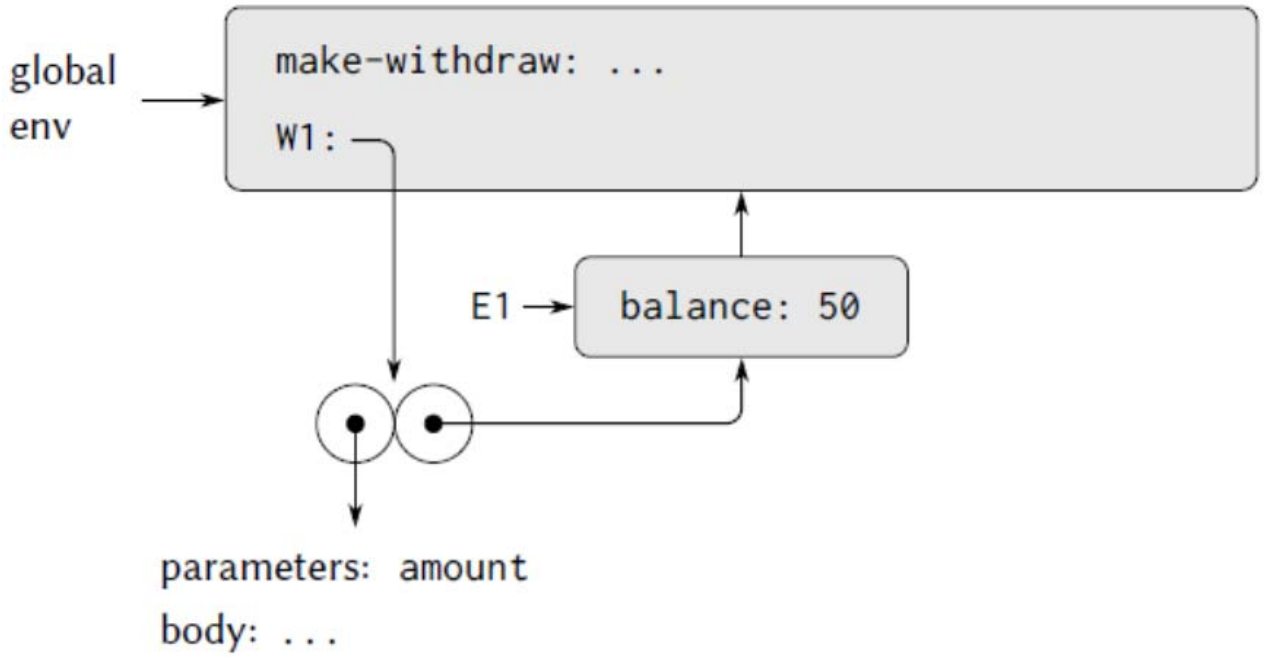
- Environment created by applying w1 to 50




```
if !balance >= amount
then begin
    balance := !balance - amount;
    !balance
end
else assert false;;
```

Frames as the Repository of Local State

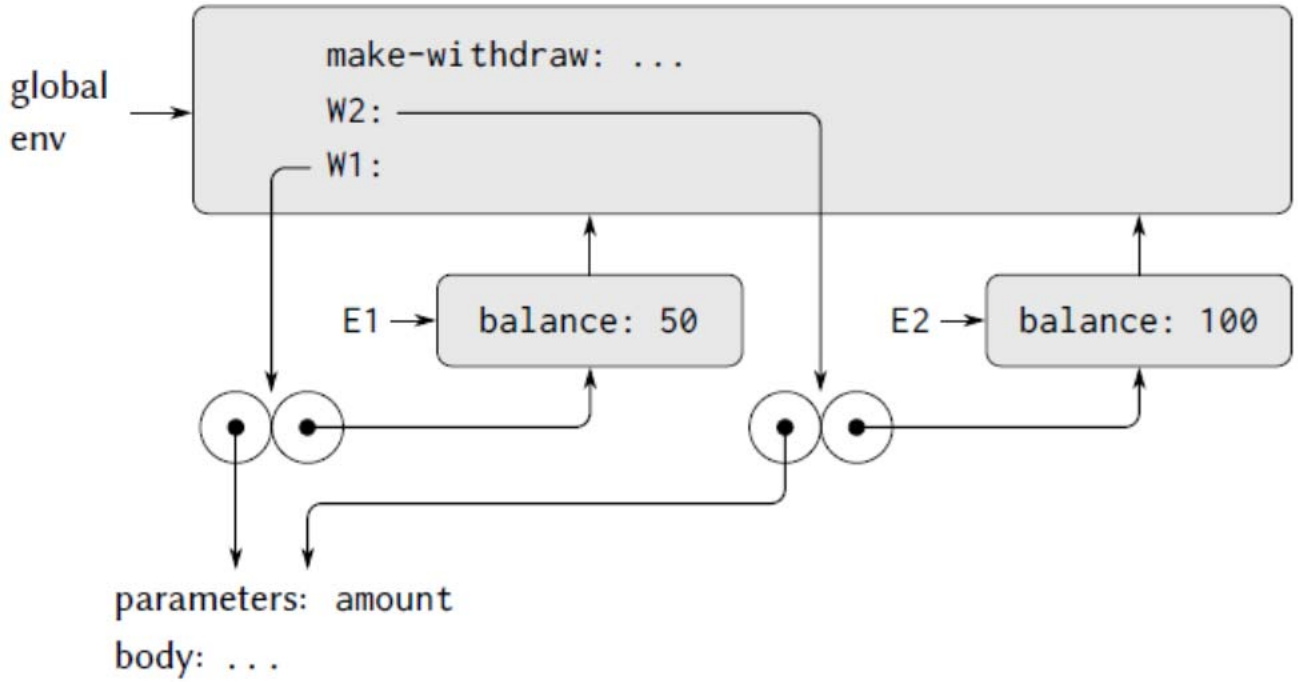
- Environment after the call to w1



Frames as the Repository of Local State

- Create a second object using

```
let w2 = make_withdraw (ref 100)
```



SPL: Evaluation

- Evaluation of SPL
  - Lookup from environment

```
let rec lookup name env =
    match env with
    | [] -> assert false
    | (n, e)::rest -> if name = n
                      then e
                      else lookup name rest
```

SPL: Evaluation

- Evaluation of SPL
  - Function definition, function application, variable

```
let rec eval expr env =
    let getClo = function Clo (v, ev, e) -> (v, ev, e)
                        | e -> print e; assert false in
…
    match expr with
…
    | V v -> lookup v env
…
    | Fun (v, e)    -> Clo (v, env, e)
    | App (f, a)    -> eval f env |> fun clo ->
                       eval a env |> fun x ->
                       getClo clo |> fun (v, ev, e) ->
                       eval e ((v, x)::ev)
…
```

SPL: Evaluation

- Testing

```
let test1 () =
    let max = Fun ("x", Fun ("y",
                If (Leq (V "x", V "y"), V "y", V "x"))) in


    let sum = Fun ("self", Fun ("x",
                    If (Equ (V "x", N 0),
                        N 0,
                        Add (V "x",
                             App (App (V "self", V "self"),
                                  Sub (V "x", N 1))))))) in
```

SPL: Evaluation

- Testing

```
    let gcd = Fun ("self", Fun ("x", Fun ("y",
                    If (Equ (V "x", V "y"),
                        V "x",
                        If (Leq (V "x", V "y"),
                            App (App (App (V "self", V "self"),
                                      Sub (V "y", V "x")),
                                      V "x"),
                            App (App (App (V "self", V "self"),
                                      Sub (V "x", V "y")),
                                      V "y")))))) in

    print (eval (App (App (max, N 1), N 2)) []);
    print (eval (App (App (sum, sum), N 10)) []);
    print (eval (App (App (App(gcd, gcd), N 12), N 20)) [])

let _ = test1 ()
```

SPL: Syntactic Sugar

```
module SyntaticSugar = struct
    let ( +  ) a b = Add (a, b)
    let ( -  ) a b = Sub (a, b)
    let ( =  ) a b = Equ (a, b)
    let ( <= ) a b = Leq (a, b)
    let ( && ) a b = And (a, b)
    let ( || ) a b = Or  (a, b)
    let ( !! ) a   = Not a
    let ( !  ) s   = V s
    let ( @  ) v e = Fun (v, e)
    let ( ->>) a b = App (a, b)
end
```

SPL: Syntactic Sugar

```
let test2 () =
    let open SyntaticSugar in

    let y = "f" @ ("k" @ !"k" ->> !"k") ->>
                  ("g" @ "x" @ !"f" ->> (!"g" ->> !"g") ->> !"x") in


    let max = "x" @ "y" @
                If (!"x" <= !"y", !"y", !"x") in


    let sum = y ->> ("sum" @ "x" @
                If (!"x" = N 0,
                    N 0,
                    !"x" + (!"sum" ->> (!"x" - N 1)))) in
```

SPL: Syntactic Sugar

```
let gcd = y ->> ("gcd" @ "x" @ "y" @
                    If (!"x" = !"y",
                        !"x",
                        If (!"x" <= !"y",
                            !"gcd" ->> (!"y" - !"x") ->> !"x",
                            !"gcd" ->> (!"x" - !"y") ->> !"y"))) in

    print (eval (max ->> N 1 ->> N 2) []);
    print (eval (sum ->> N 10) []);
    print (eval (gcd ->> N 12 ->> N 20) [])

let _ = test2()
```

SPL: Miscellaneous

```
let rec to_str e =
    let open Printf in
    match e with
    | N n -> sprintf "%d" n
    | B b -> if b then "T" else "F"
    | V v -> v
    | Add (a, b) -> sprintf "%s + %s"  (to_str a) (to_str b)
    | Sub (a, b) -> sprintf "%s - %s"  (to_str a) (to_str b)
    | Equ (a, b) -> sprintf "%s = %s"  (to_str a) (to_str b)
    | Leq (a, b) -> sprintf "%s <= %s" (to_str a) (to_str b)
    | And (a, b) -> sprintf "%s && %s" (to_str a) (to_str b)
    | Or  (a, b) -> sprintf "%s || %s" (to_str a) (to_str b)
    | Not a      -> sprintf "! %s" (to_str a)
    | If  (c, t, f) -> sprintf "(if %s then %s else %s)“
                         (to_str c) (to_str t) (to_str f)
    | Fun (v, e) -> sprintf "fun %s -> (%s)" v (to_str e)
    | Clo (v, ev, e) -> sprintf "clo %s -> (%s)" v (to_str e)
    | App (a, b) -> sprintf "%s %s" (to_str a) (to_str b)
    | _ -> assert false

let print e =
    Printf.printf "%s\n" (to_str e)
```

Parameter Passing Modes

- Terms
  - Formal parameters: parameter names in the declaration of a subroutine
  - Actual parameters (arguments): expressions that are passed to a subroutine

- Parameter passing mode
  - How the parameters are passed
  - Call by value
  - Call by reference
  - Call by name
  - Call by need

```
void square(int x) {
    x = x * x;
}

void foo() {
    square(1 + 2);
}
```

Call by Value and Call by Reference

- First, the arguments to a function are fully evaluated before invoking the function (eager evaluation)

- Call by value: copies of the arguments are passed
- Call by reference: the addresses of arguments are passed

```
void square(int x) {
    x = x * x;
}

void foo() {
    int y = 1 + 2;
    square(y);
}
```

```
void remove(Object o) {
    o = null;
}

void foo() {
    Object o = new Object();
    remove(o);
}
```

Call by Value and Call by Reference

- Why call by reference
  - To change the actual parameter value
  - When the size of actual parameter is large
- In call by value
  - Explicitly pass the addresses of variables (pointers in C)

```
void square(int* x) {
    int y = *x;
    *x = y * y;
}

void foo() {
    int y = 1 + 2;
    square( &y );
}
```

Call by Name and Call by Need

- Call by name: parameters are passed as literal substitution
  - Lazy evaluation
  - E.g. lambda calculus
- Call by need: call by name + memorize the evaluation results of actual parameters

```
int square(int x) {
    return x * x;
}

void foo() {
    square(very_complex());
}
```

```
return very_complex() *
       very_complex();
```

Lazy Evaluation

- Function application
  - Arguments (without being evaluated) are stored in an environment as a thunk
  - Delay evaluating the actual parameters until they are necessary
- When the variable is actually used
  - The thunk is forced

Lazy Evaluation

```
type expr =
…
          (*lazy* thunk: expr, env*)
          | Thk of expr * (string * expr) list
…

let eval expr env =
    let rec force = function Thk (e, ev) -> force (eval e ev)
                            | x -> x in (*lazy: if thunk force*)
    and eval expr env =
        let getNum  e = match force e with
                        N n -> n | assert false in
        let getBool e = match force e with
                        B b -> b | assert false in
        let getClo  e = match force e with
                  Clo (v, ev, e) -> (v, ev, e) | assert false in
…
```

Lazy Evaluation

```
    match expr with
    | B e -> B e
    | N n -> N n
    | V v -> lookup v env
    | Add (a, b)    -> eval a env |> fun x ->
                       eval b env |> fun y ->
                       N (getNum x + getNum y)
…
    | App (f, a)    -> eval f env |> fun clo ->
                       getClo clo |> fun (v, ev, e) ->
                       (*lazy: thunk a without evaluating it*)
                       eval e ((v, Thk (a, env))::ev)
…
```

```
let test () =
    let open SyntaticSugar in

    let y = "f" @ ("k" @ !"k" ->> !"k") ->>
                  ("g" @ !"f" ->> (!"g" ->> !"g")) in

    (*to test lazy eval*)
    let cons = "x" @ "y" @ "b" @ If (!"b", !"x", !"y") in
    let car  = "p" @ !"p" ->> B true in
    let cdr  = "p" @ !"p" ->> B false in

    let nat = y ->> ("nat" @ "n" @ (*stream of numbers from n*)
        cons ->> !"n" ->> (!"nat" ->> (!"n" + N 1))) in

    let num = y ->> ("num" @ "l" @ "n" @ (*pick n-th num in l*)
        If (!"n" = N 0,
            car ->> !"l",
            !"num" ->> (cdr ->> !"l") ->> (!"n" - N 1))) in

    (*infinite loop*)
    let sink = y ->> ("sink" @ "x" @ !"sink" ->> N 0) in

    print (eval (car ->> (cons ->> N 1 ->> (sink ->> N 0))) []);
    print (eval (num ->> (nat ->> N 0) ->> N 5) []);
```

Assignment 7

- Rewrite spl.ml in CPS
  - Download spl_cps.ml
  - Implement the TODOs in CPS
  - In CPS, the sum function should be able to handle 1000000

  - Due date: 5/26/2020

Assignment 7

- Expected output:

```
# #use "spl_cont.ml";;
type expr = …
val test1 : unit -> unit = <fun>
fun x -> (fun y -> ((if x <= y then y else x)))
fun self -> (fun x -> ((if x = 0 then 0 else x + self self x - 1)))
fun self -> (fun x -> (fun y -> ((if x = y then x else (if x <= y then
self self y - x x else self self x - y y)))))
2
55
4
…
val test2 : unit -> unit = <fun>
fun f -> (fun k -> (k k) fun g -> (fun x -> (f g g x)))
fun x -> (fun y -> ((if x <= y then y else x)))
fun f -> (fun k -> (k k) fun g -> (fun x -> (f g g x))) fun sum -> (fun x ->
((if x = 0 then 0 else x + sum x - 1)))
fun f -> (fun k -> (k k) fun g -> (fun x -> (f g g x))) fun gcd -> (fun x ->
(fun y -> ((if x = y then x else (if x <= y then gcd y - x x else gcd x - y y)))))
2
500000500000
4
```