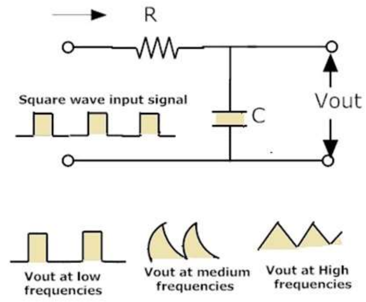
CSE216 Programming Abstractions
Objects, Streams and Lazy Evaluation

YoungMin Kwon

Modularity



- Two "*world view*" of the structure of systems: organizational strategies concentrating on
  - Objects
    - Viewing a large system as a collection of distinct objects
    - E.g. registers, inductors, capacitors, …
  - Streams
    - Flow of information in the system
    - E.g. filters, amplifiers, signal processing modules…

Assignment and Local State

- State
  - An object is said to have a state if its behavior is influenced by its history
  - E.g.) bank account: “can I withdraw $100?” depends on the history of the transition

- State variable
  - Maintain enough information about the history to determine the object’s current behavior
  - E.g.) bank account: current balance

Assignment and Local State

- Modular design
  - Decompose a model into computational objects
  - Each object has its own local state
  - Changes in the states of the objects in the system
    - $\rightarrow$ changes in the state variables of the computational object

- Assignment operator
  - Changes the value associated with a name

Mutable Data

- Arrays
  - Similar to arrays in other languages like C, Java, …

```
# let arr = [| 1; 2; 3; 4 |];;
val arr : int array = [|1; 2; 3; 4|]

# arr;;
- : int array = [|1; 2; 3; 4|]

# arr.(2);;        (* . indexing starts from 0 *)
- : int = 3

# arr.(2) <- 4;; (* <- operator for modification *)
- : unit = ()

# arr;;
- : int array = [|1; 2; 4; 4|]
```

Mutable Data

- Mutable record fields

```
type running_sum = {
    mutable sum: float;          (*mutable field*)
    mutable sum_of_squ: float;   (*sum of squares*)
    mutable count: int;
}

let mean rsum =
    rsum.sum /. float rsum.count

let variance rsum =      (*Var X = E[X^2] - E[X]^2*)
    let m_squ = rsum.sum_of_squ /. float rsum.count in
    let m = mean rsum in
    m_squ -. m *. m

let stddev rsum =
    sqrt (variance rsum)
```

```ocaml
let create () =
    { sum = 0.; sum_of_squ = 0.; count = 0 }

let update rsum x = (*update the states*)
    rsum.sum        <- rsum.sum        +. x;
    rsum.sum_of_squ <- rsum.sum_of_squ +. x *. x;
    rsum.count      <- rsum.count      +  1

let rsum = create ()

let _ = List.iter (fun x -> update rsum x) [1.;3.;2.;-7.;4.;5.]
let _ = mean rsum
let _ = stddev rsum
```

```
# #use "running_sum.ml";;
…
- : float = 1.33333333333333
- : float = 3.94405318873307
```

Mutable Data

- Refs
  - *record type* with a *single mutable field* called *contents*

```
# let x = ref 0;; (* create a ref, i.e., {contents = 0}*)
val x : int ref = {contents = 0}

# x;;
- : int ref = {contents = 0}

# !x;;             (* ! get the contents of a ref, i.e. x.contents *)
- : int = 0

# x := !x + 1;;    (* := assignment, i.e., x.contents <- … *)
- : unit = ()

# x.contents <- x.contents + 1;;
- : unit = ()

# x;;
- : int ref = {contents = 2}
```

Local State Variable

- Computational object with time varying state
  - E.g.) balance of a bank account: each invocation of withdraw returns a different balance
    - If the initial balance was $100
    - `withdraw 10` returns 90
    - `withdraw 10` returns 80, ...
  - The same `withdraw 10` returns different values
  - To implement withdraw, we can use a variable balance
    - balance decrements by the amount of withdraw

Local State Variable

```
# let balance = ref 100;; (*time varying state variable*)
val balance : int ref = {contents = 100}


# let withdraw amount =
    if !balance >= amount (*balance is outside of withdraw*)
    then begin
        balance := !balance - amount; (*balance changes*)
        !balance
    end
    else
        assert false;;
val withdraw : int -> int = <fun>


# withdraw 10;;
- : int = 90

# withdraw 10;;
- : int = 80
```

Local State Variable

```
# let make_withdraw balance = (*balance is a local state var*)
    fun amount ->
        if !balance >= amount
        then begin
            balance := !balance - amount;
            !balance
        end
        else assert false;;
val make_withdraw : int ref -> int -> int = <fun>


# let withdraw = make_withdraw (ref 100);;
val withdraw : int -> int = <fun>


# withdraw 10;;
- : int = 90

# withdraw 10;;
- : int = 80
```

SUNY Korea

Local State Variable

- Computational object with local states
  - ⇒ assignments with local variables

- Problem: the substitution model does not work
  - Substitution model: replace the function parameters variables with actual parameter expressions

```
let double x =
    x + x

double (withdraw 10);;
- : int = 180
```

```
In substitution model

double (withdraw 10)
=> (withdraw 10) + (withdraw 10)
=> - : int = 170
```

Bank Account Example

```
(*bank account object
*)
type action = Withdraw | Deposit

let make_account balance =
    let bal = ref balance in  (*bal is a local state var*)

    let withdraw amount =     (*subtract amount from bal*)
        if !bal >= amount
        then begin
            bal := !bal - amount;
            !bal
        end
        else
            assert false in

    let deposit amount =      (*add amount to bal*)
        bal := !bal + amount;
        !bal in
…
```

Bank Account Example

```
…
    let dispatch msg =   (*dispatch: message-passing style*)
        match msg with
        | Withdraw -> withdraw
        | Deposit  -> deposit in

    dispatch (*return dispatch as a result*);;

val make_account : int -> action -> int -> int = <fun>


# let acc = make_account 100;;
val acc : action -> int -> int = <fun>

# acc Withdraw 10;;
- : int = 90

# acc Deposit 20;;
- : int = 110
```

Benefits of Introducing Assignments

- Modular design
  - Viewing systems as a collection of objects with local state
  - Without local variables, modularity can be broken
  - E.g.) Monte Carlo simulation
    - $6/\pi^2$ is equal to the probability that two randomly chosen integers will have no common factors
    - $6/\pi^2$ ~ the probability of `gcd (rand ()) (rand ()) = 1`

```ocaml
(* Estimating pi: Monte Carlo simulation on Cesaro test
   local state: x in rand
*)

let rand_update x = (x * 16807) mod 0x7fffffff

let make_rand rand_init =
    let x = ref rand_init in
    fun () ->
        x := rand_update !x;
        !x

let rand = make_rand 1

let rec gcd a b =
    if      a = 0 then b
    else if b = 0 then a
    else if a > b then gcd (a mod b) b
    else               gcd (b mod a) a

let cesaro_test () =  (*experiment*)
    (gcd (rand ()) (rand ())) = 1
```

```
(*monte_carlo: estimates the probability that an experiment succeeds
    - the function can be used for different experiments
*)
let monte_carlo trials experiment =
    let rec iter n cnt_passed =
        if n = 0 then
            (float cnt_passed) /. (float trials)
        else if experiment () then
            iter (n - 1) (cnt_passed + 1)
        else
            iter (n - 1) (cnt_passed)     in
    iter trials 0


let estimate_pi trials =
    sqrt (6. /. (monte_carlo trials cesaro_test))

let _ = estimate_pi 1000000

# #use "montecarlo.ml";;
- : float = 3.1410943510648726
```

Benefits of Introducing Assignments

- Monte Carlo simulation without using a local state
  - Use rand_update instead of rand
  - Monte Carlo idea cannot be isolated
    - experiment cannot be passed as a parameter
    - monte_carlo method is fixed for the cesaro test
  - The state variable x for rand should be carried through the iter function

```ocaml
(*Monte Carlo simulation WITHOUT a local state
  monte_carlo cannot be separated from the experiment (cesaro test)
*)
let monte_carlo trials  =
    let rec iter n cnt_passed r =
        let r1 = rand_update r  in
        let r2 = rand_update r1 in
        if n = 0 then
            (float cnt_passed) /. (float trials)
        else if (gcd r1 r2) = 1 then
            iter (n - 1) (cnt_passed + 1) r2
        else
            iter (n - 1) (cnt_passed)     r2 in
    iter trials 0 1


let estimate_pi trials =
    sqrt (6. /. (monte_carlo trials))

let _ = estimate_pi 1000000
```

Costs of Introducing Assignments

- The substitution model does not work
- “Nice” mathematical properties cannot be an adequate framework for objects
  - Referential transparency (equals can be substituted for equals) is violated
  - Reasoning about programs becomes drastically more difficult

```
# let w1 = make_withdraw (ref 25);;
val w1 : int -> int = <fun>

# let w2 = make_withdraw (ref 25);;
val w2 : int -> int = <fun>
```

```
# w1 10;;
- : int = 15

# w1 10;;
- : int = 5
# w2 10;;
- : int = 15
```

SUNY Korea

Costs of Introducing Assignments

- Functional programming
  - Programming without using assignments
  - Procedures can be viewed as mathematical functions
    - Two evaluations of the same procedure with the same arguments produce the same result
  - Referential transparency is preserved

Costs of Introducing Assignments

- Imperative programming
  - Extensive use of assignments
  - The order of assignment is important
  - E.g.) factorial

```
let factorial n =
    let rec iter p c =
        if c > n then p
        else iter (c * p)
                  (c + 1) in
    iter 1 1 in
```

```
let factorial2 n =
    let p = ref 1 in
    let c = ref 1 in
    let rec iter () =
        if !c > n then !p
        else begin
          p := !c * !p;
          c := !c + 1;
          iter ()
        end in
    iter ()
```

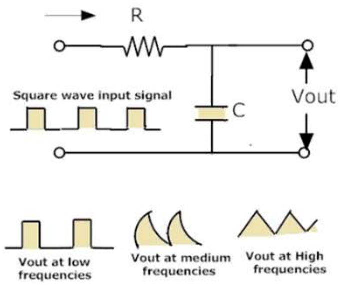
Their order is important

Streams

- Modeling state
  - We saw assignments as a tool for modeling states
    - Real world object with local state → computational object with local variable
    - Time variation in the real world → assignments to local variables
  - Streams: alternative approach
    - Time varying behavior of a variable x → a function of time $x(t)$
    - The function itself does not change

Streams

- Streams
  - A stream is simply a sequence
  - Delayed evaluation → enables representing very large (possibly infinite) sequences as a stream
  - Streams → modeling systems with states without using assignments



Abstractions for Sequences

- *Abstractions* for manipulating sequences
  - *`map`*, *`filter`*, *`accumulate`*, …
  - Elegantly manipulate sequences
  - But the elegance is bought at the price of inefficiency (both time and space)

Abstractions for Sequences

```
(*sum of prime numbers in the interval [a, b]
*)
let sum_primes1 a b =
    let iter count accum =
        if count > b then
             accum
        else if isPrime count then
            iter (count + 1) (count + accum)
        else
            iter (count + 1) accum in
    iter a 0
```

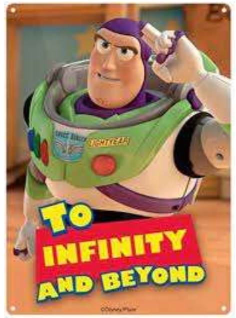
- sum_primes1 needs to store only the accumulated sum

Abstractions for Sequences

```
(*sum of prime numbers in the interval [a, b]
*)                                       [a; a+1; a+2; … b]
let sum_primes2 a b =
    enumerate_interval a b |> fun i ->
    filter isPrime i       |> fun p ->
    accumulate (+) 0 p
```

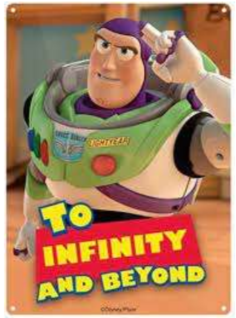
- sum_primes2: generates an interval of list, generates a filtered list, then accumulate the list

Streams are Delayed Lists



- With streams
  - Formulate programs elegantly as a sequence manipulation
  - Attaining the efficiency of incremental computation
- Streams
  - Construct a stream only partially
  - Pass the partial construction to its consumer
  - If the consumer tries to access unconstructed part of the stream → the stream will construct just enough more

Streams



- stream type, car, cdr

```
type 'a stream = Nil
               | Cons of 'a * (unit -> 'a stream) (*thunk*)

let cons a thunk =
    Cons (a, thunk)

let car s =
    match s with
    | Nil -> assert false
    | Cons (x, _) -> x

let cdr s =
    match s with
    | Nil -> assert false
    | Cons (_, f) -> f () (*force*)


let rec from n = cons n (fun () -> from (n + 1))
let nat = from 0 (*natural numbers*)
```

Streams

- map and filter functions

thinking will delay
further evaluation

```
let rec map proc s =
    if s = Nil then Nil
    else cons (proc (car s))
              (fun () -> map proc (cdr s))


let plus1 = map (fun x -> x + 1) nat

let _ = plus1 |> cdr |> cdr |> cdr |> car

- : int = 4
```

Streams

- map and filter functions

```
let rec filter predi s =
    if s = Nil then Nil
    else if predi (car s) then
        cons (car s)
             (fun () -> filter predi (cdr s))
    else filter predi (cdr s)

let even = filter (fun x -> x mod 2 = 0) nat

let _ = even |> cdr |> cdr |> cdr |> car


- : int = 6
```

thunking will delay further evaluation

Fibonacci Numbers with Streams

- Fibonacci numbers

```
let fibs =
    let rec fibgen a b =
        cons a (fun () -> fibgen b (a + b)) in
    fibgen 0 1

let _ = fibs |> cdr |> cdr |> cdr |> cdr |> cdr |> car

- : int = 5
```

Prime Numbers with Streams

- Prime numbers (*sieve of Eratosthenes*)
  - Start with 2 (the 1st prime number)
  - Get streams by filtering the multiples of 2
    - Leaves us a stream string with 3 (the 2nd prime number)
  - Get streams by filtering the multiples of 3
    - Leaves us a stream string with 5 (the 3rd prime number)
  - Get streams by filtering the multiples of 5
    - Leaves us a stream string with 7 (the 4th prime number)
  - …

Prime Numbers with Streams

- Prime numbers by *sieve of Eratosthenes*

```
let rec sieve s =
    let h = car s in
    let thunk = fun () ->
        cdr s
        |> filter (fun x -> (x mod h) <> 0)
        |> sieve in
    cons h thunk

let primes = from 2 |> sieve

(*n-th element of s*)
let rec stream_ref n s =
    if n = 0 then car s
             else stream_ref (n - 1) (cdr s)

let _ = stream_ref 50 primes
```

Monte Carlo Simulation with Streams

- Monte Carlo simulation using a stream
  - make_rand returns a stream of pseudo random numbers
  - cesaro_stream is a stream of Cesaro experiment
  - Monte Carlo simulation is separated from experiment
    - monte_carlo runs Cesaro test passed as an experiment
    - Modularity is regained

Monte Carlo Simulation with Streams

- Stream of random numbers

```
let rand_update x = (x * 16807) mod 0x7fffffff

let rec make_rand rand_init =
    let next = rand_update rand_init in
    cons next
         (fun() -> make_rand next)
```

rand is a stream of random numbers

- Stream of Cesaro test results

```
let rec cesaro_stream rand =
    let first  = car rand in
    let rest   = cdr rand in
    let second = car rest in
    cons (gcd first second = 1)
         (fun() -> cesaro_stream (cdr rest))
```

experiment is a stream of cesaro tests

Monte Carlo Simulation with Streams

- Monte Carlo simulation

monte_carlo is a stream
of probabilities

```
let rec monte_carlo passed trials experiment =
    let next passed trials =
        let h = (float passed) /. (float trials) in
        let thunk = fun() ->
            monte_carlo passed trials (cdr experiment) in
        cons h thunk in
    if car experiment
    then next (passed + 1) (trials + 1)
    else next passed (trials + 1)
```

Monte Carlo Simulation with Stream

- The simulation program

```
let pi =  make_rand 1
       |> cesaro_stream
       |> monte_carlo 0 0
       |> map (fun p -> sqrt (6. /. p))

(*n-th element of s*)
let rec stream_ref n s =
    if n = 0
    then car s
    else cdr s |> stream_ref (n -1)

let _ = pi |> stream_ref 100000
```

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);
}
```

SUNY Korea

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

Tiny: Lazy Evaluation

- Call-by-name examples

```
(*no division by 0 error*)
(   (lambda (c t f)
         (if c t f))
    true (+ 1 1) (/ 1 0))

(*stream of natural numbers*)
(   (lambda (rec cons car cdr)
         (    (lambda (s) (car (cdr (cdr (cdr s))))) (*4th element*)
              (rec (lambda (self n)                   (*natural numbers*)
                      (cons n (self self (+ n 1))))   (*stream wo thunk*)
                 0)))
    (lambda (f) (f f))
    (lambda (x y z) (if z x y))
    (lambda (x) (x true))
    (lambda (x) (x false)))
```

Tiny: Lazy Evaluation

```
type expr
    = NUM  of int    (*number*)
    | BOOL of bool   (*Boolean*)
    | VAR  of string (*variable*)
    (*arithmetic exprs*)
    | ADD of expr * expr | SUB of expr * expr
…
    (*function definition: parameter, body*)
    | FUN of string * expr
    (*closure: parameter, body, environment*)
    | CLO of string * expr * (string * expr) list

    (*lazy eval, thunk: expr, env*)
    | TNK of expr * (string * expr) list

    (*function application: operator, operand*)
    | APP of expr * expr
```

```
(*evaluate expr in env*)
let rec eval expr env =
…
    let force = function
            | TNK (e, ev) -> eval e ev (*lazy*)
            | x -> x in

    match expr with
    | BOOL b -> BOOL b
    | NUM n  -> NUM n
    | VAR v  -> lookup v env |> force
…
    | FUN (v, e)    -> CLO (v, e, env)

    (*lazy: thunk a and env without evaluating a*)
    | APP (f, a)    -> eval f env   |> fun clo ->
                       dropCLO clo |> fun (v, e, ev) ->
                       eval e ((v, TNK (a, env))::ev)

    | _ -> assert false
```

```
(*no division by 0 error*)
let ite = parse
            "(  (lambda (c t f)\
                    (if c t f))\
                true (+ 1 1) (/ 1 0))"

(*stream without thunking*)
let nat = parse
            "(  (lambda (rec cons car cdr)\
                    (   (lambda (s) (car (cdr (cdr (cdr s))))))\ (*4th*)
                        (rec (lambda (self n)\ (*nat: stream wo thunk*)
                                (cons n (self self (+ n 1)))))\
                             0)))\
                (lambda (f) (f f))\
                (lambda (x y z) (if z x y))\
                (lambda (x) (x true))\
                (lambda (x) (x false)))"

eval ite [] |> print;
eval nat [] |> print;
()
```

Results:
NUM(2)
NUM(3)

Optional: Lazy module of OCaml

- Lazy module
  - lazy <expr>: make expr of u type u Lazy.t type without evaluating it
  - Lazy.force <expr>: evaluate Lazy.t type expr

```
module type Lazy = sig
    type 'a t = 'a lazy_t
    val force: 'a t -> 'a
end
```

```
# let a = lazy (1+2);;
val a : int lazy_t = <lazy>

# Lazy.force a;;
- : int = 3
```

Optional: Lazy module of OCaml

- Lazy stream

```
module Stream = struct
    type 'a stream = Nil |
    Cons of 'a * (unit -> 'a stream)

    let cons a thunk =
        Cons (a, thunk)

    let car s =
        match s with
        | Cons (x, _) -> x
        | _ -> assert false

    let cdr s =
        match s with
        | Cons (_, f) -> f ()
        | _ -> assert false
end
```

```
module StreamLazy = struct
    type 'a stream = Nil |
    Cons of 'a * 'a stream Lazy.t

    let cons a thunk =
        Cons (a, thunk)

    let car s =
        match s with
        | Cons (x, _) -> x
        | _ -> assert false

    let cdr s =
        match s with
        | Cons (_, f) -> Lazy.force f
        | _ -> assert false
end
```

Optional: Lazy module of OCaml

```ocaml
module StreamTest = struct
    open Stream

    let rec sum a b =
        cons ((car a) + (car b))
        (fun () -> sum (cdr a) (cdr b))

    let rec fibs () =
        cons 1 (fun () ->
        cons 2 (fun () ->
        sum (fibs ()) (cdr (fibs ()))
        ))

    let rec stream_ref s n =
        if n = 0
        then car s
        else stream_ref (cdr s) (n -1)
end

let _ = let open StreamTest in
        stream_ref (fibs ()) 10
```

```ocaml
module StreamLazyTest = struct
    open StreamLazy

    let rec sum a b =
        cons ((car a) + (car b))
        (lazy (sum (cdr a) (cdr b)))

    let rec fibs () =
        cons 1 (lazy (
        cons 2 (lazy (
        sum (fibs ()) (cdr (fibs ()))
        ))))

    let rec stream_ref s n =
        if n = 0
        then car s
        else stream_ref (cdr s) (n -1)
end

let _ = let open StreamLazyTest in
        stream_ref (fibs ()) 10
```

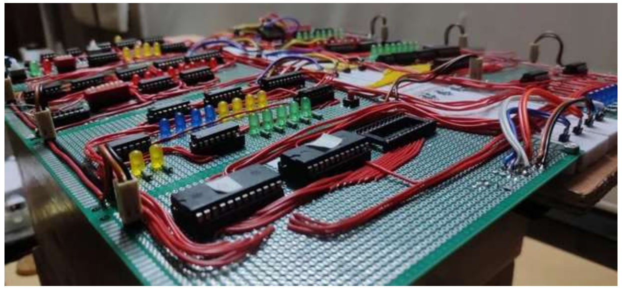
SUNY Korea

Assignment 6

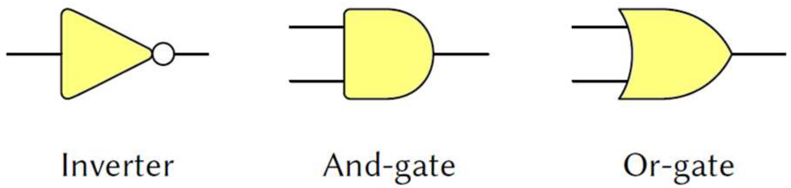
- In this assignment we will simulate simple circuits using streams
  - Wires as a stream
  - Logical gates
  - Half adder, Full adder, and n-bit adder
  - Download, adder.ml; implement all TODOs; and submit adder.ml to Brightspace
- Due date 5/2/2024

```
(** Stream ****************************
*)
module type IStream = sig
    type 'a stream = Nil | Cons of 'a * (unit -> 'a stream)

    val cons: 'a -> (unit -> 'a stream) -> 'a stream
    val nil: unit -> 'a stream
    val car: 'a stream -> 'a
    val cdr: 'a stream -> 'a stream
    val index: 'a stream -> int -> 'a
end

(* TODO: impelemnt Stream module
*)
module Stream: IStream = struct
    type 'a stream = Nil | Cons of 'a * (unit -> 'a stream)

    let cons h t =
    let nil () =
    let car = function
    let cdr = function
    let rec index s n =     (*return the n-th element of stream s*)
end
```

```
(** Wire ****************************
*)
module type IWire = sig
    include IStream
    type wire = int stream

    val w_zero: wire
    val w_one: wire
    val probe: wire list -> int -> unit
end

module Wire: IWire = struct
    include Stream
    type wire = int stream

    (* TODO: implement constant
       constant c returns the infinite stream of c
    *)
    let rec constant c =


    let w_zero = constant 0
    let w_one  = constant 1
…
end
```



```
(** Gate ****************************
*)
module type IGate = sig
    open Wire

    val g_not: wire -> wire
    val g_and: wire -> wire -> wire
    val g_or:  wire -> wire -> wire
end

module GateBuilder (P: IGateParam): IGate = struct
    open Wire

    (* delay d stream adds d 0's to the front of stream
        e.g. delay 3 stream => [0; 0; 0; stream]
    *)
    let rec delay d stream =
        if d = 0
        then stream
        else cons 0 (fun () -> delay (d-1) stream)
```

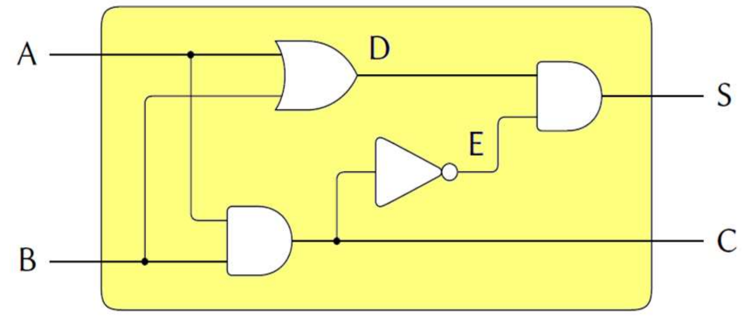


…

```
(*not-gate: returns the negated stream of w_a
    e.g. g_not [1; 1; 0; 0; ...] => [0; 0; 1; 1; ...]
*)
let g_not w_a =
    let rec iter wa =
        let a = car wa in
        let o = if a = 0 then 1 else 0 in
        cons o (fun () -> iter (cdr wa)) in
    iter w_a |> delay P.delay_not

(*TODO: impement g_and, the and-gate
    - g_and returns the stream of the conjunction of w_a and w_b
    e.g. g_and [1; 1; 0; 0; ...] [1; 0; 1; 0; ...] => [1; 0; 0; 0; ...]
*)
let g_and w_a w_b =


(*TODO: impement g_or, the or-gate
    - g_or returns the stream of the disjunction of w_a and w_b
    e.g. g_and [1; 1; 0; 0; ...] [1; 0; 1; 0; ...] => [1; 1; 1; 0; ...]
*)
let g_or w_a w_b =
```

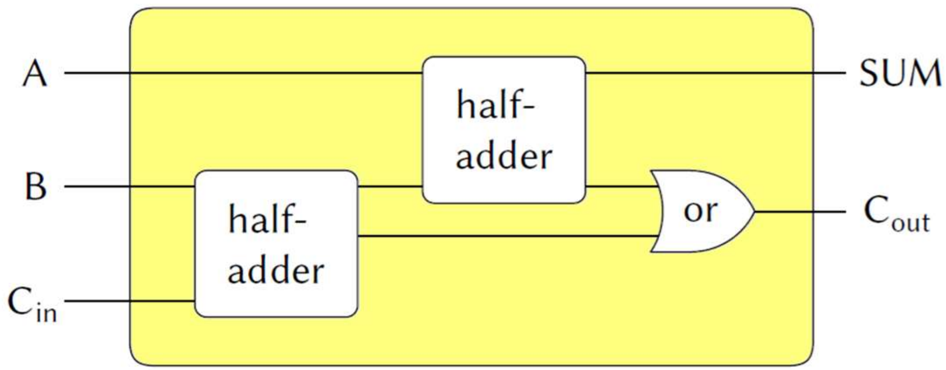
```
end
```

```
(** Adder ****************************
*)
module type IAdder = sig
    open Wire

    val half_adder: wire -> wire -> (wire * wire)
    val full_adder: wire -> wire -> wire -> (wire * wire)
    val adder:      wire list -> wire list -> wire list
end

module AdderBuilder (G:IGate) : IAdder = struct
    open Wire
    open G

    (*TODO: impement half_adder, a half-adder
        - half_adder returns the tuple of the sum and the carry streams
          of w_a and w_b
        e.g. half_adder [1; 1; 0; 0; ...]
                        [1; 0; 1; 0; ...]
                    => ([0; 1; 1; 0; ...], [1; 0; 0; 0; ...])
    *)
    let half_adder w_a w_b =
```



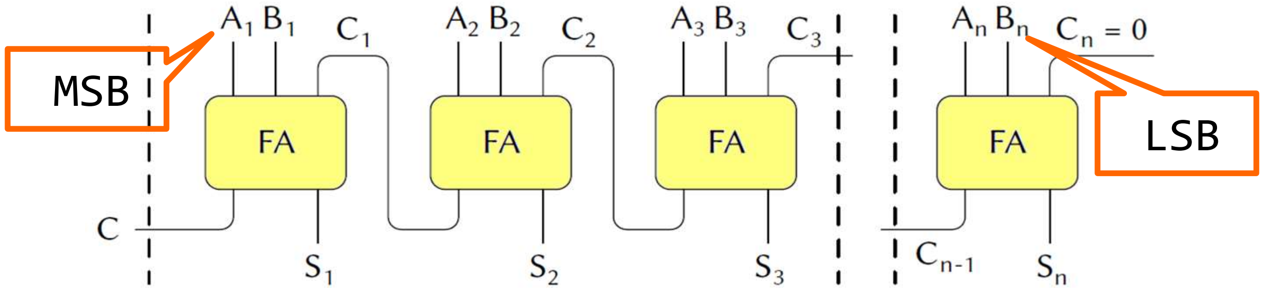
A half-adder circuit.

```
(*TODO: impement full_adder, a full-adder
    - full_adder returns the tuple of the sum and the carry streams
      of w_a, w_b, and w_c
    e.g. half_adder [1; 1; 0; 0; ...]
                    [1; 0; 1; 0; ...]
                    [1; 1; 0; 0; ...]
                => ([1; 0; 1; 0; ...], [1; 1; 0; 0; ...])
*)
let full_adder w_a w_b w_c =
```



A full-adder circuit.

```
(*TODO: impement adder, an n-bit adder
    - wl_a, wl_b: list of wires of the form [LSB wire; ... ; MSB wire]
    - adder returns the sum of wl_a, wl_b with carry in the form
      [LSB wire; ... ; MSB wire; carry wire]
    e.g. adder [  [1; 0; 1; 1; ...];
                  [1; 1; 1; 0; ...]  ]
               [  [1; 1; 1; 0; ...];
                  [0; 1; 1; 1; ...]  ]
        =>     [  [0; 1; 0; 1; ...];
                  [0; 0; 1; 1; ...];
                  [1; 1; 1; 0; ...]  ]

       i.e. adder [3; 2; 3; 1; ...]
                  [1; 3; 3; 2; ...]
               => [(0, 1); (1, 1); (2, 1); (3, 0); ...]
*)
let adder wl_a wl_b =
    let rec iter wl_a wl_b w_c =

    iter wl_a wl_b w_zero
end
```



A ripple-carry adder for n-bit numbers.