Haskell and Skeleton Libraries

- Haskell is a general purpose, functional programming language;
- Library for perfectly synchronous process networks;
- Library for data flow process networks;

Basic Haskell Features

- *Declarative*: The program is a list of equations which have to be satisfied; no predefined execution order;
- *Single assignment*: “Variables” are assigned only once;
- *Higher order functions*: Functions are first class citizens and can be used as arguments;
- *Partial evaluation*
- *Strong type system*

Haskell Example

Quicksort in Haskell:

```
qsort :: [Int] -> [Int]
qsort [ ]      = [ ]
qsort (x:xs) = (qsort lt_x) ++ [x] ++ (qsort greq_x)
           where
             lt_x     = [y | y <- xs, y < x]
             greq_x = [y | y <- xs, y >= x]
```


Data and Types

Examples:

```
5  :: Integer
'a' :: Char
inc :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
```

Build-in types:

Int	- integer
Integer	- integer
Bool	- boolean (True, False)
Float	- floating point
String	- string of characters
Char	- character

Function Types

- Many functions are polymorphic, i.e. they operate on different types:

```
length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + length xs

head :: [a] -> a
head (x:xs) = x
tail :: [a] -> [a]
tail (x:xs) = xs
```


Partial Evaluation of Functions

```
add :: Integer -> Integer -> Integer
add x y = x+y

inc :: Integer -> Integer
inc = add 1

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

```
mymap = map inc
mymap :: [Integer] -> [Integer]
mymap [1,2,3]            => [2,3,4]

monitor :: Show a => String -> a -> a
monitorF1 = monitor “file1”
```


User Defined Types

- Type synonyms:

```
type String = [Char]
type IntPair = (Int,Int)
```

- Type declarations:

```
data Bool = False | True
data Color = Red | Green | Blue deriving (Show,Eq)
data Tval a = Abst
            | Prst a deriving (Eq, Show)
data Signal a = NullS
              | a :- Signal a
              deriving (Eq, Show)
```


Lists, Strings, and Tuples

```
l1 = [1,2,3] ++ [4,5,6]      => [1,2,3,4,5,6]
l2 = 1 : [2,3,4,5,6]         => [1,2,3,4,5,6]
st1 = [‘a’,’b’,’c’] ++ “efg”  => “abcefg”
s1 :: Signal Int
s1 = signal [1,2,3,4]
s2 = s1 +: s1                => 1 :- 2 :- 3 :- 4 :- 1 :- 2 :- 3 :- 4 :- NullS
s3 = 0 :- s1                 => 0 :- 1 :- 2 :- 3 :- 4 :- NullS
s4 :: Signal (Int,Int)
s4 = signal [(1,2), (3,4)]   => (1,2) :- (3,4) :- NullS
type TInt = TVal Int
ts1 :: Signal Tint
ts1 = signal [Abst, Prst 1, Prst 2]   => Abst :- (Prst 1) :- (Prst 2) :- NullS
```

Infix Operators

- List concatenation:

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

- Function composition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

```
mapSY :: (a -> b) -> Signal a -> Signal b
p1 = mapSY inc
s1 = signal [1,2,3]
p1 s1            => 2 :- 3 :- 4 :- NullS

p2 = mapSY (*2)
p2 s1            => 2 :- 4 :- 6 :- NullS

p3 = p2 . p1
p3 s1            => 4 :- 6 :- 8 :- NullS
```


Lazy Evaluation

- Only values which are needed are computed.

```
il1 = [1,2..]
il2 = [1,3..]
take 5 il1          => [1,2,3,4,5]
take 3 il2          => [1,3,5]
take 5 (drop 5 il1) => [6,7,8,9,10]
s1 = signal [1..]
takeS 3 s1          => 1 :- 2 :- 3 :- NullS
```

```
p1 = mapSY inc
p2 = mapSY (*2)
p3 = p2 . p1
takeS 5 (p3 s1)
   => 4 :- 6 :- 8 :- 10 :- 12 :- NullS
takeS 3 (drop 98 (p3 s1))
   => 200 :- 202 :- 204 :- NullS
```


Pattern Matching

```
take 0 _       = []
take _ []      = []
take n (x:xs) = x : take (n-1) xs
```

- *Guards*:

```
sign x | x > 0   = 1
       | x== 0   = 0
       | x < 0   = -1
sign x | x > 0       = 1
       | x== 0       = 0
       | otherwise   = -1
```

- *As-patterns*

```
f (x:xs)   = x : x : xs

f s@(x:xs)  = x : s
```

- Wild cards:

```
head (x:_)   = x
tail   (_:xs)  = xs
```

- *Case expressions:*

```
take m ys = case (m,ys) of
              (0,_)      -> []
              (_,[])     -> []
              (n,x:xs) -> x : take (n-1) xs
```


Synchronous Library - State-less Skeletons

```
mapSY :: (a -> b) -> Signal a -> Signal b
p1 = mapSY (*2)
p1 (1:-2:-3:-NullS)        => 2 :- 4 :- 6 :- NullS
zipWithSY :: (a -> b -> c) -> Signal a -> Signal b -> Signal c
p2 = zipWithSY (*)
p2 (1:-2:-3:-NullS) (1:-2:-3:-NullS)  => 1 :- 4 :- 9 :- NullS
zipWith3SY :: (a -> b -> c -> d) -> Signal a -> Signal b -> Signal c -> Signal d
zipWith4SY :: (a -> b -> c -> d -> e)
   -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e
zipSY :: Signal a -> Signal b -> Signal (a,b)
zipSY (1:-2:-3:-NullS) (4:-5:-6:-NullS)   => (1,4):-(2,5):-(3,6):-NullS)
unzipSY :: Signal (a,b) -> (Signal a, Signal b)
unzipSY (zipSY (1:-2:-3:-NullS) (4:-5:-6:-NullS))
   =>  ((1:-2:-3:-NullS), (4:-5:-6:-NullS))
```

Synchronous Library - State Skeletons

```
scanlSY :: (a -> b -> c) -> a -> Signal b -> Signal a
scanlSY f mem NullS = NullS
scanlSY f mem (x:-xs) = f mem x :- (scanlSY f newmem xs)
                    where newmem = f mem x
scanl2SY :: (a -> b -> c -> d) -> a -> Signal b -> Signal c -> Signal a
```

```
cntrFn :: Int -> Bool -> Int
cntrFn y x =
      case (x) of
      True   -> ((y+1) `mod` 4)
      False  -> y
```

```
p1 = scanlSY cntrFn 0

s1 = signal (concat (repeat [True,False]))
s2 = p1 s1

takeS 7 s2   => 1 :- 1 :- 2 :- 2 :- 3 :- 3
                :- 0 :- NullS
```


Synchronous Library - State Machines

```
mooreSY :: (a -> b -> c) -> (a -> c) -> a -> Signal b -> Signal a
mooreSY nextstate output init = mapSY output . (scanlSY nextstate init)
mealySY :: (a -> b -> c) -> (a -> b -> c) -> a -> Signal b -> Signal a
mooreSY nextstate output init signal
   = zipWithSY output (init :- (scanlSY nextstate init signal)) signal
```

```
outFn :: Int -> String
outFn 0 = "zero"
outFn 1 = "one"
outFn 2 = "two"
outFn 3 = "three"
```

```
p2 = mealySY cntrFn outFn 0
s3 = p2 s1
take 7 s3
   => "one" :- "one" :- "two" :- "two"
      :- "three" :- "three" :- "zero"
      :- NullS
```


Synchronous Library - Timed Signals

```
type Value = Integer
type ValEvent = TVal Value

s1 :: Signal ValEvent
s1 = signal [Abst, (Prst 1),
             Abst, (Prst 2)]

data Header = UHd | EHd1 | EHd2
type Body = [Int]
type Packet = (Header,Body)
type PacketEv = TVal Packet
```

```
hdTrans :: PacketEv -> PacketEv
hdTrans Abs = Abs
hdTrans p@((Prst UHd),_) = p
hdTrans ((Prst EHd1),b) = ((Prst EHd2),b)
hdTrans ((Prst EHd2),b) = ((Prst EHd1),b)

p = mapSY hdTrans

s1 = (Abst :- (Prst (UHd, [0,1]))
   :-(Prst (EHd1,[])) :- (Prst (EHd2,[])))

p s1  => Abst :- (Prst (UHd, [0,1]))
   :-(Prst (EHd2,[])) :- (Prst (EHd1,[]))
```


Data Flow Library

- Data flow skeletons do not operate on timed signals.
- Data flow skeletons allow combinatorial functions which consume several input and output tokens.

```
mapDF      ::        (a -> b)        -> Signal a -> Signal b
map12DF    ::        (a -> (b,b))    -> Signal a -> Signal b
map21DF    ::        ((a,a) -> b)    -> Signal a -> Signal b
map22DF    ::        ((a,a) -> (b,b)) -> Signal a -> Signal b
mapGen     :: Int -> ([a] -> [b])    -> Signal a -> Signal b
mapUpDF    ::        (a -> [b])      -> Signal a -> Signal b
mapDownDF  :: Int -> ([a] -> b)      -> Signal a -> Signal b
```

Stochastic Processes and Skeletons

- sigmaUn generates a signal of uniformly distributed integers within a given range.

```
sigmaUn :: Int -> (Int, Int) -> Signal Int
takeS 5 (sigmaUn 3 (0,9))        => 5 :- 5 :- 4 :- 7 :- 3 :- NullS
```

- selMapDF is a stochastic variant of mapDF;
  it selects one out of two functions to be applied on the input based on a probability distribution.

```
selMapDF :: Int -> (a -> b) -> (a -> b) -> Signal a -> Signal b

selScanlDF :: Int -> (a -> b -> a) -> (a -> b -> a) -> a -> Signal b -> Signal a
```


How to Model with Skeletons ?

1. Draw a process graph.
2. Decide the types of processes:
   no internal state -> map skeleton
   internal state same as output -> scanl skeleton
   output depends only on state -> moore skeleton
   output depends on input and state -> mealy
3. Define data types of signals and states.
4. Decide on skeleton variants (# inputs, # outputs,…)
5. Design the combinatorial functions

How to Model with Skeletons ?

- 6. Compose the processes into a system.

```
p1 :: Signal a -> Signal b -> Signal c
p2 :: Signal c -> (Signal d, Signal e)
p3 :: Signal e -> Signal b

system sa = sd
        where   sd = fst (p2 sc)
                sc = p1 sa sb
                sb = p3 se
                se = snd (p2 sc)
```