韩冬@滴滴FP

CPS TRANSFORM

CPS -- CONTINUATION PASSING STYLE

- continuation 指广义 (:: a->b) 上的后续计算。
- continuation passing style 指使用 continuation (:: a -> b)和接受 continuation (:: (a -> b) -> c) 的函数来构建计算的一种编程思路。

```
"I'm a fox" :: String
```

$$\Big\downarrow \text{CPS}$$

```
\ c -> c "I'm a fox" :: (String -> r) -> r

(\ c -> c "I'm a fox") id -- "I'm a fox"
```

CPS和代数类型一样，可以用来作为控制结构的基础。

```
data Bool = True | False
if :: Bool -> a -> a -> a
if True x _ = x
if False _ y = y

checkFoo :: Bool -> String
checkFoo b = if b "Foo" "Bar"

type Bool = forall a. a -> a -> a
true :: Bool
true = \ x -> \ _ -> x
false :: Bool
false = \ _ -> \ y -> y

checkFoo :: Bool -> String
checkFoo b = b "Foo" "Bar"
```

常见的CPS套路

```
data Position = Position Pico Pico
p = Position 3 4 :: Position
type PositionCPS = forall r. (Pico -> Pico -> r) -> r
pCPS k = k 3 4    :: Position

data Either a b = Left a | Right b
type EitherCPS a b = forall r. (a -> r) -> (b -> r) -> r

error = left "i'll be back"     :: Either String b
errorCPS k _ = k "i'll be back" :: EitherCPS String b

data Maybe a = Just a | Nothing
type MaybeCPS a = forall r. (a -> r) -> r -> r
JustCPSFoo sk _ = sk "Foo"
nothingCPS _ fk = fk
```

CPS和MONAD

```
instance Monad Maybe where
    return x = Just x
    Just x  >>= f = f x
    Nothing >>= _ = Nothing

envPort = Just "8080"
readInt :: String -> Maybe Int
foo :: Int -> Maybe Foo

envPort >>= readInt >>= foo
case (case envPort of
           Nothing -> Nothing
           Just foo  -> readInt foo
      ) of
    Nothing -> Nothing
    Just i  -> foo i
```

CPS和MONAD

```
newtype MaybeCPS r a = MaybeCPS
    { runMaybeCPS :: (a -> r) -> r -> r }

instance Monad (MaybeCPS r) where
    return x = MaybeCPS (\ks kf -> ks x)
    MaybeCPS c >>= f = MaybeCPS $
        \ ks kf ->          <———  (b -> r) -> r -> r
           (c $ \ a ->
              runMaybeCPS (f a) ks kf) kf
```

b -> r

(a -> r) -> r -> r

a -> MaybeCPS b

(b -> r) -> r -> r

CPS和MONAD

```
envPort = MaybeCPS $ \ ks kf -> kf
readInt str = MaybeCPS $ \ ks kf -> ...

envPort >>= readInt >>= foo
(MaybeCPS $
    \ ks' kf' ->
        ((\ks kf -> kf) $ \ a ->
            runMaybeCPS (readInt a) ks' kf') kf'
) >>= foo

MaybeCPS $ \ ks'' kf'' -> ((\ ks' kf' -> ... ) $ ...)
runMaybeCPS s f

-- MaybeCPS $ \ ks kf -> kf
```

CONT MONAD

```
newtype Cont r a = Cont
    { runContT :: (a -> r) -> r }

instance Monad (Cont r) where
    return x = Cont $ \ k -> k x
    Cont c >>= f = Cont $ \ k ->
        c $ \ a ->
            runContT (f a) k
```

```
b -> r
```

```
(a -> r) -> r
```

```
a -> Cont r b
```

```
(b -> r) -> r
```

CONT MONAD

```
envPortCPS :: Cont r String
envPortCPS = Cont $ \ k -> k "8080"

readIntCPS :: String -> Cont r Int
readIntCPS str = Cont $ \ k -> k (readInt str)

fooCPS :: Cont r Int
fooCPS = do
    port <- envPortCPS
    readIntCPS port

fooCPS foo
-- foo (readInt envPort)
```

CALL-WITH-CURRENT-CONTINUATION

```
callCC :: ((a -> Cont r b) -> Cont r a)
       -> Cont r a
callCC f = Cont $ \ a ->
    runContT $ f (\ x -> Cont $ \ _ -> a x)

fooCPS :: Cont r Int
fooCPS = do
    port <- envPortCPS
    err <- callCC $ \ k -> do
        port_num <- readInt port
        if port_num < 80 then k "not available"
                         else ...
    catchErr err
```

REALWORLD EXAMPLE: ATTOPARSEC

```
type Failure i t r = t -> Pos
                       -> More
                       -> [String]
                       -> String
                       -> IResult i r

type Success i t a r = t -> Pos
                         -> More
                         -> a
                         -> IResult i r

newtype Parser = Parser { runParser :: forall r. Input
                                    -> Pos
                                    -> More
                                    -> Failure i Input r
                                    -> Success i Input a r
                                    -> IResult i r
```

REALWORLD EXAMPLE: ATTOPARSEC

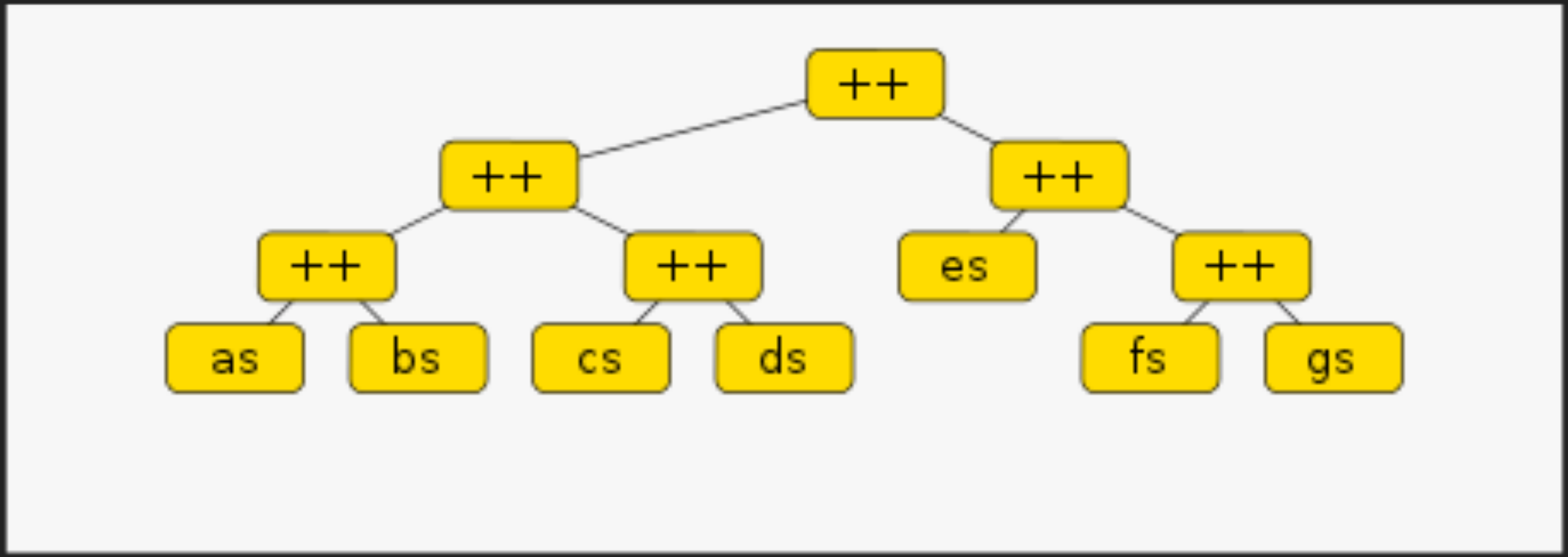
```
instance Monad (Parser i) where
    return v = Parser $ \t pos more _lose succ ->
                                   succ t pos more v

    m >>= k = Parser $ \t !pos more lose succ ->
        let succ' t' !pos' more' a =
                runParser (k a) t' pos' more' lose succ
        in runParser m t pos more lose succ'
```

COMPOSE CONTINUATION

```
type ShowS = String -> String

shows "hello" -- (hello ++)
shows 'X'     -- ('X':)

newtype DList a = DL { unDL :: [a] -> [a] }

append :: DList a -> DList a -> DList a
append xs ys = DL (unDL xs . unDL ys)
```

LIST

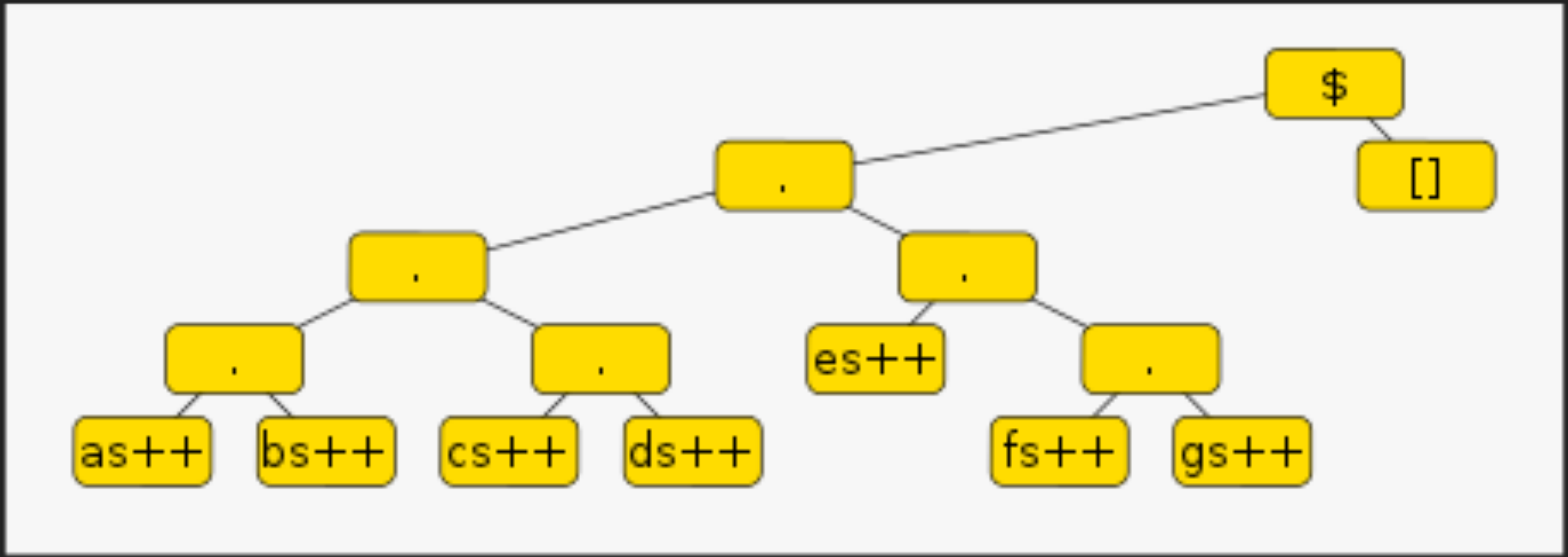
```
l = (((as ++ bs) ++ (cs ++ ds))
    ++ (es ++ (fs ++ gs)))

xs ++ ys = case xs of []        -> ys
                      (x:xs') -> x : (xs' ++ ys)
```



```
process (x:xs) = ...
process l
```

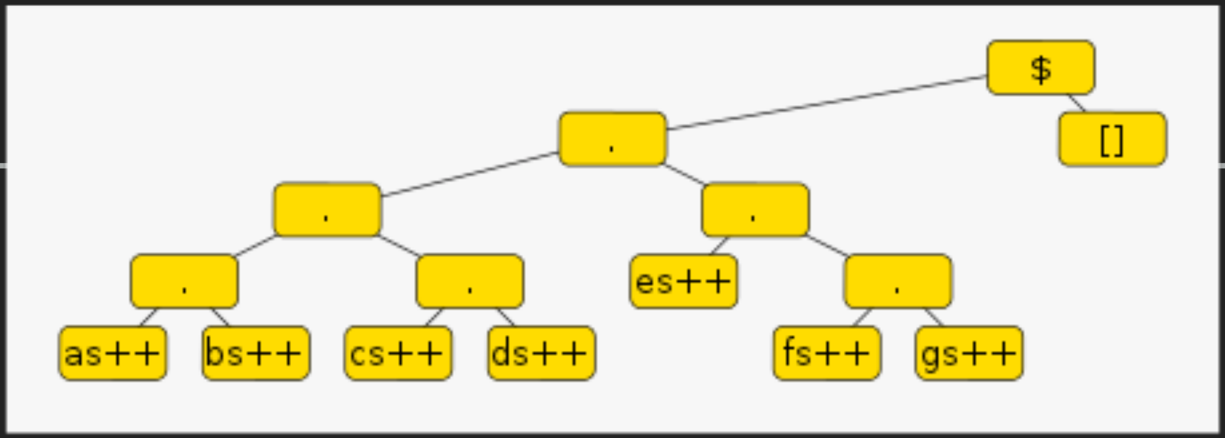
DLIST

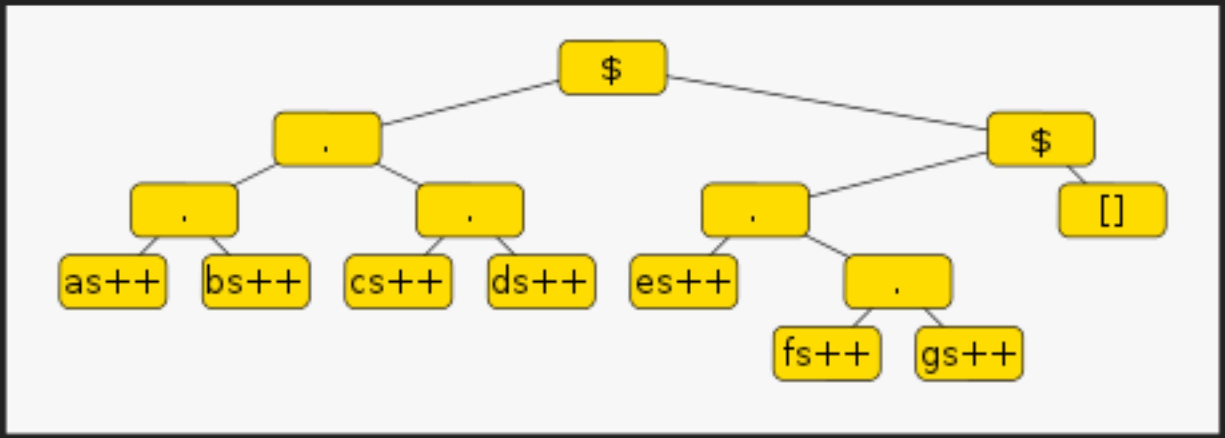
```
dl = ((((as++) . (bs++)) . ((cs++) . (ds++))) .
     ((es++) . ((fs++) . (gs++))))

dl []
```

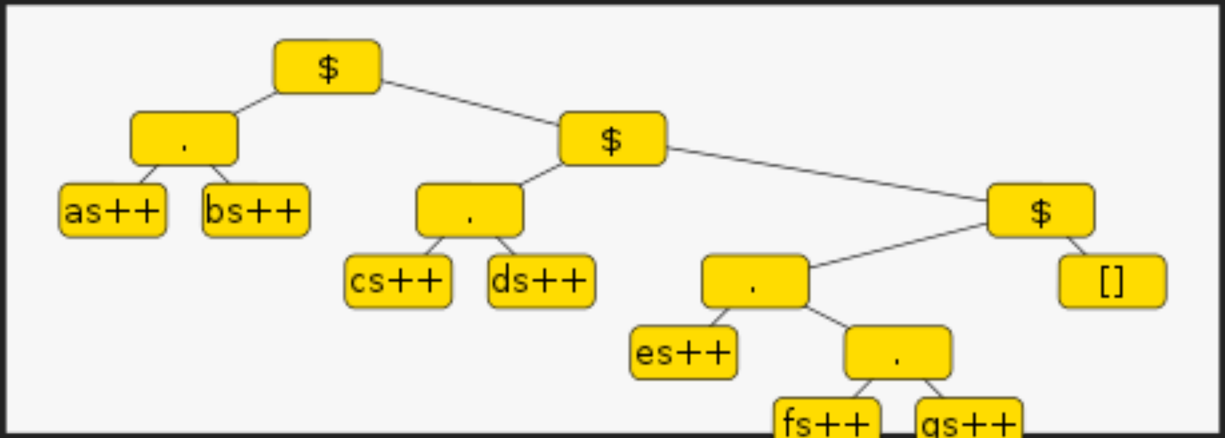


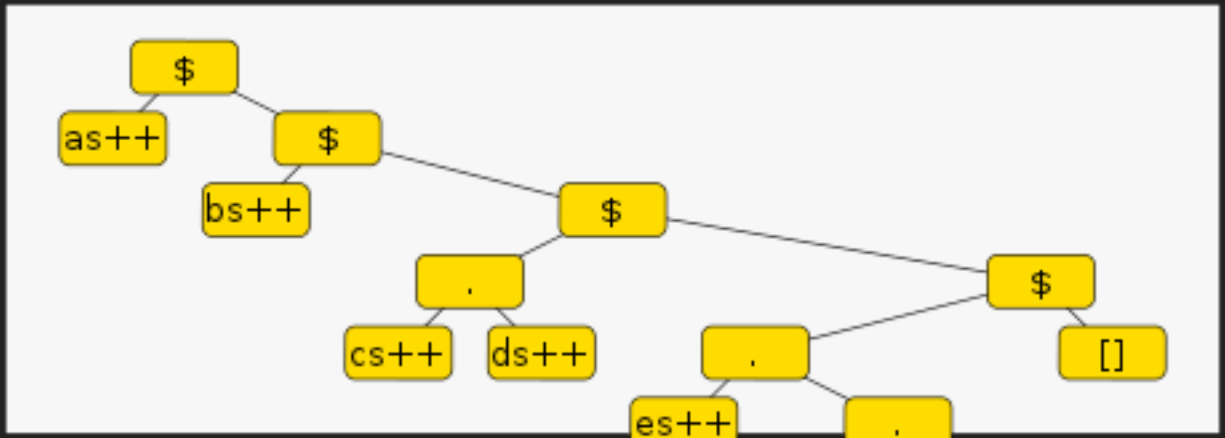
DLIST

```
(as++
  (bs++
    (cs++
      (ds++
        (es++
          (fs++
            (gs++[])))))))

process (x:xs) = ...
process l
```









FORMAT TIME

```
data Part = Year | Month | Day | Hour | Min | Sec | Str String
              deriving (Show, Eq)

class FormatTime t where
    formatPart :: t -> [Part] -> [Part]

data Date = Date
    { year :: Int
    , month :: Int
    , day :: Int
    }

data Clock = Clock
    { hour :: Int
    , min  :: Int
    , second :: Int
    }
```

FORMAT TIME

```
instance FormatTime Date where
    -- formatPart :: Date -> [Part] -> [Part]
    formatPart dt@(Date y m d) = map go
      where
        go p =
            case p of Year -> Str (show y)
                      Month -> Str (show m)
                      Day -> Str (show d)
                      _   -> p

formatPart (Date 2016 12 27)
           [ Month, Str "-", Day, Str " "
           , Hour, Str ":", Min]
-- [ Str "12", Str "-", Str "27", Str " "
-- , Hour, Str ":", Min]
```

FORMAT TIME

```
instance FormatTime Clock where
    -- formatPart :: Clock -> [Part] -> [Part]
    formatPart c@(Clock h m s) = map go
      where
        go p =
            case p of Hour -> Str (show h)
                      Min -> Str (show m)
                      Sec -> Str (show s)
                      _   -> p

formatPart (Clock 20 43 00)
           [ Month, Str "-", Day, Str " "
           , Hour, Str ":", Min]
-- [ Month, Str "-", Day, Str " "
-- , Str "20", Str ":", Str "43"]
```

FORMAT TIME

```
data DateTime = DateTime Date Clock

instance FormatTime DateTime where
    -- formatPart :: DateTime -> [Part] -> [Part]
    formatPart (DateTime d c) =
        formatPart d . formatPart c

formatPart
    (DateTime (Date 2016 12 27) (Clock 20 43 00))
    [ Month, Str "-", Day, Str " "
    , Hour, Str ":", Min]

--  [ Str "12", Str "-", Str "27", Str " "
--  , Str "20", Str ":", Str "43"]
```

FORMAT TIME

```haskell
formatTime :: (FormatTime t) => t -> [Part] -> String
formatTime t ps = go (formatPart t ps)
  where
    go []       = ""
    go (p:ps) = case p of Str s -> s ++ go ps
                          _     -> go ps

formatTime
    (DateTime (Date 2016 12 27) (Clock 20 43 00))
    [ Month, Str "-", Day, Str " "
    , Hour, Str ":", Min]

--  [ Str "12", Str "-", Str "27", Str " "
--  , Str "20", Str ":", Str "43"]

--  "12-27 20:43"
```

FORMAT TIME CPS

```
data Part = Year | Month | Day | Hour | Min
          | Sec   | Str String
    deriving (Show, Eq)

class FormatTime t where
    formatPart :: t
               -> (Part -> String)
               -> Part -> String
```

THE CONTINUATION K!

FORMAT TIME CPS

```
instance FormatTime Date where
    -- formatPart :: Date
    --            -> (Part -> String)
    --            -> Part -> String

    formatPart (Date y m d) k p =
        case p of Year  -> show y
                  Month -> show m
                  Day   -> show d
                  Str x -> x
                  _     -> k p
```

THE CONTINUATION K!

FORMAT TIME CPS

```
instance FormatTime Clock where
    -- formatPart :: Clock
    --            -> (Part -> String)
    --            -> Part -> String

    formatPart (Clock h m s) k p =
        case p of Hour -> show h
                  Min  -> show m
                  Sec  -> show s
                  Str x -> x
                  _    -> k p
```

FORMAT TIME CPS

```
instance FormatTime DateTime where
    -- formatPart :: DateTime
    --            -> (Part -> String)
    --            -> Part -> String

    formatPart (DateTime d c) k =
        formatPart d (formatPart c k)

formatTime :: (FormatTime t) => t -> [Part] -> String
formatTime t ps =
    concat $ map (formatPart t (const "")) ps
```