

Efektid ja monaadid

Efektid ja monaadid

Sissejuhatus

- Monaadid on abstraktsioon kõrvalefektidega arvutuste esitamiseks "puhtas maailmas".
- Pärinevad kategooriateooriast.
- Moggi (1988) kasutas monaade denotatsioonsemantika struktureerimiseks.
- Wadler (1990) näitas, et analoogselt on võimalik monaade kasutada efektidega funktsionaalprogrammide struktureerimiseks.
- Haskell 1.3 (1996) võttis monaadid kasutusele IO struktureerimiseks.
- ...

Kategoriateooria põhidefinitsioonid

Kategooriad

Kategooria $\mathcal{C}$ koosneb:

- kollektsioonist **objektidest** $\text{Obj}(\mathcal{C})$;
- iga $A, B \in \text{Obj}(\mathcal{C})$ jaoks, kollektsoonist **morfismidest** (ehk **nooltest**) $\text{Mor}(A, B)$;
 - morfismi $f \in \text{Mor}(A, B)$ tähistatakse $f : A \rightarrow B$
- iga $A \in \text{Obj}(\mathcal{C})$, **identsusmorfism** $\text{id}_A : A \rightarrow A$;
- iga $f : A \rightarrow B$ ja $g : B \rightarrow C$ korral leidub morfism $g \cdot f : A \rightarrow C$ (f ja g **kompositsioon**);
- identsused ja kompositsioon rahuldavad võrdusi:

$$f = f \cdot \text{id}_A \quad f = \text{id}_B \cdot f \quad h \cdot (g \cdot f) = (h \cdot g) \cdot f$$

kus $f : A \rightarrow B$, $g : B \rightarrow C$ ja $h : C \rightarrow D$.

Kategoriateooria põhidefinitsioonid

Näited

Hulkade kategooria Set :

- objektid on hulgad;
- morfismid on kujutised hulkade vahel.

Osaliste funktsioonide kategooria Pfn :

- objektid on hulgad;
- morfismid on osalised kujutised hulkade vahel.

Osaliste funktsioonide kategooria Rel :

- objektid on hulgad;
- morfismid on relatsioonid hulkade vahel.

Osaliste funktsioonide kategooria Mon :

- objektid on monoidid;
- morfismid on monoidide homomorfismid.

Kategoorieooria põhidefinitsioonid

Funktorid

Olgu $\mathcal{C}$, $\mathcal{D}$ kategooriad. **Funktor** $F : \mathcal{C} \rightarrow \mathcal{D}$ koosneb kahest kujutisest:

- kujutis objektidel $F : \text{Obj}(\mathcal{A}) \rightarrow \text{Obj}(\mathcal{B})$, ja
- kujutis morfismidel:

$$\begin{aligned} F(f : A \rightarrow B) & : F A \rightarrow F B \\ F(id_A) & = id_{FA} \\ F(g \cdot f) & = F(g) \cdot F(f) \end{aligned}$$

Funktorit $F : \mathcal{C} \rightarrow \mathcal{C}$ (so. mille lähte- ning sihtkategooria on samad) nimetatakse **endofunktoriks**.

Kategooriateooria põhidefinitsioonid

Näited

- Identsus- ja konstantne funktor:

$$\begin{aligned}\text{Id} : \mathcal{C} &\rightarrow \mathcal{C} \\ X &\mapsto X \\ f &\mapsto f\end{aligned}$$

$$\begin{aligned}\text{K}_A : \mathcal{C} &\rightarrow \mathcal{D} \\ X &\mapsto A \\ f &\mapsto \text{id}_A\end{aligned}$$

- Unustav- ja "powerset"-funktor:

$$\begin{aligned}\text{U} : \text{Mon} &\rightarrow \text{Set} \\ X &\mapsto X \\ f &\mapsto f\end{aligned}$$

$$\begin{aligned}\text{P} : \text{Set} &\rightarrow \text{Set} \\ \text{P}(X) &= \{Y \mid Y \subseteq X\} \\ \text{P}(f) &= X \mapsto \{f\,x \mid x \in X\}\end{aligned}$$

- Listide funktor $\text{List} : \text{Set} \rightarrow \text{Mon}$ (aga ka $\text{List} : \text{Set} \rightarrow \text{Set}$)

$$\begin{aligned}\text{List}(X) &= X^* \\ \text{List}(f) &= [x_1, \dots, x_n] \mapsto [f\,x_1, \dots, f\,x_n]\end{aligned}$$

Kategooriateooria põhidefinitsioonid

Loomulikud teisendused

Olgu $F, G : \mathcal{C} \rightarrow \mathcal{D}$ funktorid. **Loomulik teisendus** $\tau : F \rightarrow G$ on morfismide pere $\tau_X : FX \rightarrow GX$ selline, et iga morfismi $f : X \rightarrow Y$ korral, järgmine ruut kommuteerub:

$$\begin{array}{ccc} FX & \xrightarrow{\tau_X} & GX \\ Ff \downarrow & & \downarrow Gf \\ FY & \xrightarrow{\tau_Y} & GY \end{array}$$

Kategoriateooria põhidefinitsioonid

Näited

- Listide pööramine:

$$\text{reverse} : \text{List} \rightarrow \text{List}$$

$$\text{reverse} [x_1, \dots, x_n] = [x_n, \dots, x_1]$$

$$\text{reverse} \cdot \text{List}(f) = \text{List}(f) \cdot \text{reverse}$$

- Listi pikkuse leidmine:

$$\text{length} : \text{List} \rightarrow \mathbb{K}_{\mathbb{N}}$$

$$\text{length} [x_1, \dots, x_n] = n$$

$$\begin{aligned} \text{length} \cdot \text{List}(f) &= \mathbb{K}_{\mathbb{N}}(f) \cdot \text{length} \\ &= \text{length} \end{aligned}$$

Kategoriateooria põhidefinitsioonid

Kategooriad ja Haskell

Haskelli võime mõelda kategooriana, kus:

- objektideks on (monomorfsed) tüübid;
- morfismideks on (monomorfsed) funktsioonid;
- (endo-)funktoriteks on tüübikonstruktorid (ehk polümorfsed tüübid);
- loomulikeks teisendusteks on (parameetriliselt) polümorfsed funktsioonid.

NB!

Tüübikonstruktorid teevad tüüpidest tüüpe, ehk siis on teisendus objektide vahel. Funktor peab teisendama ka morfismid morfismideks.

Kategooriad ja Haskell

Klass *Functor*

```
class Functor f where  
    fmap :: (a → b) → f a → f b
```

NB!

- Vastavalt funktori definitsioonile, peab *fmap* lisaks veel säilitama identsust ja kompositsiooni.
- Seda Haskellis kompilaator ei kontrolli, ning see on puhtalt programmeerija vastutusel, et *fmap* tõepoolest seda teeb.

Kategooriad ja Haskell

Listide funktor

```
instance Functor [] where  
    fmap = map
```

Maybe funktor

```
data Maybe a = Nothing | Just a  
instance Functor Maybe where  
    fmap f Nothing = Nothing  
    fmap f (Just x) = Just (f x)
```

Aritmeetilste avaldiste väärtustamine

Aritmeetilste avaldised

```
data Expr = Num Int
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr *: Expr
          | Expr :/: Expr
```

Näited

```
exp1 = Num 1 :+: Num 2
exp2 = Num 2 *: (Num 4 :-: Num 1)
exp3 = Num 4 *: Num 3 :/: Num 2
```

Aritmeetiliste avaldiste väärtustamine

Lihtne väärtustaja

$eval :: Expr \rightarrow Int$

$eval (Num\ i) = i$

$eval (e1\ +:\ e2) = (eval\ e1) + (eval\ e2)$

$eval (e1\ -:\ e2) = (eval\ e1) - (eval\ e2)$

$eval (e1\ *:\ e2) = (eval\ e1) * (eval\ e2)$

$eval (e1\ :/\ e2) = (eval\ e1)\ 'div'\ (eval\ e2)$

Näited

Main> *eval exp2*

6

Main> *eval (Num 3 :/\ Num 0)*

Program error: divide by zero

Aritmeetilste avaldiste väärtustamine

Veatöõtlusega väärtustaja

$eval :: Expr \rightarrow Maybe Int$

$eval (Num\ i) = Just\ i$

$eval (e1\ :+: e2) = \text{case } eval\ e1 \text{ of}$
 $Nothing \rightarrow Nothing$
 $Just\ v1 \rightarrow \text{case } eval\ e2 \text{ of}$
 $Nothing \rightarrow Nothing$
 $Just\ v2 \rightarrow Just\ (v1 + v2)$

$eval (e1\ :-: e2) = \dots$

$eval (e1\ *: e2) = \dots$

Aritmeetiliste avaldiste väärtustamine

Veatöötusega väärtustaja (järg)

```
eval (e1 :/: e2) = case eval e1 of
    Nothing → Nothing
    Just v1 → case eval e2 of
        Nothing → Nothing
        Just v2 → if v2 ≡ 0
                    then Nothing
                    else Just (v1 'div' v2)
```

Näited

```
Main> eval exp2
Just 6
Main> eval (Num 3 :/: Num 0)
Nothing
```

Aritmeetilste avaldiste väärtustamine

NB!

- Suur hulk koodi duplitseerimist!
- Mis on ühtne muster mida välja faktoriseerida?
- Paneme tähele, et:
 - toimub alamavaldiste väärtustamiste järjestikustamine;
 - kui üks ebaõnnestub, siis ebaõnnestub ka kogu avaldise väärtustamine;
 - väärtustamise õnnestumisel tehakse resultaat kättesaadavaks järgnevatele väärtustustele.

Aritmeetiliste avaldiste väärtustamine

Väärtustuste järjestikustamine

```
evSeq :: Maybe Int → (Int → Maybe Int) → Maybe Int
evSeq ma f = case ma of
    Nothing → Nothing
    Just a   → f a
```

Aritmeetilste avaldiste väärtustamine

Veatöötusega väärtustaja

$eval :: Expr \rightarrow Maybe Int$

$eval (Num\ i) = Just\ i$

$eval (e1\ :+:\ e2) = eval\ e1\ 'evSeq'\ \lambda v1 \rightarrow$
 $eval\ e2\ 'evSeq'\ \lambda v2 \rightarrow$
 $Just\ (v1 + v2)$

$eval (e1\ :-:\ e2) = \dots$

$eval (e1\ :*\ e2) = \dots$

$eval (e1\ :/\ e2) = eval\ e1\ 'evSeq'\ \lambda v1 \rightarrow$
 $eval\ e2\ 'evSeq'\ \lambda v2 \rightarrow$
if $v2 \equiv 0$
then *Nothing*
else $Just\ (v1\ 'div'\ v2)$

Ebaõnnestumistega arvutused

Üldistus

- Tüübist *Maybe a* võime mõelda kui tüüpi a väärtusi väljastavatest **arvutustest**, mis võivad ka **ebaõnnestuda**; so.
 - arvutus võib õnnestuda, ning väljastab tulemusena tüüpi a väärtuse;
 - arvutus võib ebaõnnestuda, ning tulemust ei väljastata;
 - arvutusi saab järjestikku komponeerida, so. üksteise järel täita.
- Järjestikkompositsiooni korral ühe alamarvutuse ebaõnnestumine põhjustab kogu arvutuse ebaõnnestumise.
- Seega, ebaõnnestumine on **efekt**, mõjutades kaudselt kõiki järgnevaid arvutusi.

Ebaõnnestumistega arvutused

Ebaõnnestumistega arvutuste monaad

$mReturn \quad :: a \rightarrow Maybe\ a$

$mReturn\ a \quad = Just\ a$

$mFail \quad :: Maybe\ a$

$mFail \quad = Nothing$

$mBind \quad :: Maybe\ a \rightarrow (a \rightarrow Maybe\ b) \rightarrow Maybe\ b$

$mBind\ ma\ f = \text{case } ma \text{ of}$

$\quad Nothing \rightarrow Nothing$

$\quad Just\ a \rightarrow f\ a$

Aritmeetilste avaldiste väärtustamine

Veatöõtlusega väärtustaja

$eval :: Expr \rightarrow Maybe Int$

$eval (Num\ i) = mReturn\ i$

$eval (e1\ :+:\ e2) = eval\ e1\ 'mBind'\ \lambda v1 \rightarrow$
 $eval\ e2\ 'mBind'\ \lambda v2 \rightarrow$
 $mReturn\ (v1 + v2)$

$eval (e1\ :-:\ e2) = \dots$

$eval (e1\ :*\ e2) = \dots$

$eval (e1\ :/\ e2) = eval\ e1\ 'mBind'\ \lambda v1 \rightarrow$
 $eval\ e2\ 'mBind'\ \lambda v2 \rightarrow$
 $if\ v2 \equiv 0$
 $\quad then\ mFail$
 $\quad else\ mReturn\ (v1\ 'div'\ v2)$

Puu lehtede nummerdamine

Binaarsed "lehtpuud"

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
```

Näited

```
tree1 = Branch (Leaf 7) (Leaf 3)
tree2 = Branch (Branch (Leaf 'A')
                  (Branch (Leaf 'B')
                          (Leaf 'C'))))
        (Branch (Leaf 'D')
                (Leaf 'E'))
```

Puu lehtede nummerdamine

Lehtede nummerdamine

$labelTree \quad :: \quad Tree \ a \rightarrow Tree \ Int$

$labelTree \ t = fst \ (labAux \ t \ 0)$

$labAux \ :: \ Tree \ a \rightarrow Int \rightarrow (Tree \ Int, Int)$

$labAux \ (Leaf \ x) \quad \quad \quad c = (Leaf \ c, c + 1)$

$labAux \ (Branch \ t1 \ t2) \ c = \text{let } (t1', c1) = labAux \ t1 \ c$
 $\quad \quad \quad (t2', c2) = labAux \ t2 \ c1$
 $\quad \quad \quad \text{in } (Branch \ t1' \ t2', c2)$

Puu lehtede nummerdamine

NB!

- Loendur algul intsialiseeritakse ning lehtedes kasutatakse tema hetkeväärtust.
- Alampuude korrektseks nummerdamiseks tuleb loendurit vastavates arvutustes kaasa tassida ning erinevate alam arvutuste vahel korrektses järjekorras edastada.
 - On väga vigade altis, kuna väga lihtne on kogemata edastada loendurist vale versioon.
 - Imperatiivkeeltes võiksime loendurina kasutada globaalset muutujat, mille edastamine alam arvutustele toimub varjatult.

Olekust sõltuvad arvutused

Üldistus

- **Olekust sõltuv arvutus** kasutab väärtuse arvutamiseks mingit olekut ning arvutuse käigus võib muuta ka olekut.
- Olekust sõltuvat arvutust saame esitada **olekuteisendajana**, so. funktsioonina mis argumendina saab sisendoleku ning väljastab resultaadi koos väljundolekuga.
 - Arvutus võib mitte vajada olekut, misjuhul väljundolek on sama mis sisendolek ning resultaat on olekust sõltumatu.
 - Olekust sõltuvate arvutuste järjestikku komponeerimisel tuleb eelneva alamarvutuse väljundolek anda järgneva arvutuse sisendolekuks.
 - Ainult olekut manipuleerivad olekuteisendajad on efektiivsed.

Olekust sõltuvad arvutused

Olekuteisndajad

```
type IntSt a = Int → (a, Int)
sReturn      :: a → IntSt a
sReturn x    = λs → (x, s)
inc          :: IntSt Int
inc          = λs → (s, s + 1)
sBind        :: IntSt a → (a → IntSt b) → IntSt b
m 'sBind' f  = λs → let (x1, s1) = m s
                  (x2, s2) = f x1 s1
                  in (x2, s2)
```

Puu lehtede nummerdamine

Lehtede nummerdamine

$$labelTree \quad :: \quad Tree \ a \rightarrow Tree \ Int$$
$$labelTree\ t = fst\ (labAux\ t\ 0)$$
$$labAux \quad :: \quad Tree \, a \rightarrow IntSt \, (Tree \, Int)$$
$$labAux (Leaf\ x) = inc\ 'sBind'\ \lambda i \rightarrow sReturn\ (Leaf\ i)$$
$$\begin{aligned} \text{labAux } (\text{Branch } t1 \ t2) &= \text{labAux } t1 \text{ 'sBind' } \lambda t1' \rightarrow \\ &\quad \text{labAux } t2 \text{ 'sBind' } \lambda t2' \rightarrow \\ &\quad \text{sReturn } (\text{Branch } t1' \ t2') \end{aligned}$$

Monaadid

Võrdlus

- Mõlema näite korral on põhiomaduseks efektidega arvutuste teostamine.
- Mõlemal juhul sai efektidega arvutusi selgelt esitada tuues sisse identse struktuuriga järgmised abstraktsioonid:
 - tüüp, mis esitab arvutusi;
 - kombinaator, mis esitab ilma efektideta väärtusi;
 - kombinaator arvutuste järjestikku komponeerimiseks.
- Vastava struktuuriga arvutused ongi **monaadid**.

Monaadid

Definitsioon

Monaad kategoorial $\mathcal{C}$ koosneb:

- endofunktorist $T : \mathcal{C} \rightarrow \mathcal{C}$
- loomulikust teisendusest $\eta_A : A \rightarrow TA$
 - nn. monaadi **ühik**
- loomulikust teisendusest $\mu_A : TTA \rightarrow TA$
 - nn. monaadi **multiplikatsioon**

selliselt, et järgnevad diagrammid kommuteeruvad:

$$\begin{array}{ccccc} T^2A & \xleftarrow{T\eta_A} & TA & \xrightarrow{\eta_{TA}} & T^2A \\ & \searrow \mu_A & \parallel & \swarrow \mu_A & \\ & & TA & & \end{array}$$

$$\begin{array}{ccc} T^3A & \xrightarrow{\mu_{TA}} & T^2A \\ T\mu_A \downarrow & & \downarrow \mu_A \\ T^2A & \xrightarrow{\mu_A} & TA \end{array}$$

Monaadid

Definitsioon

Kleisli kolmik kategoorial $\mathcal{C}$ koosneb:

- kujutisest $T : \text{Obj}(\mathcal{C}) \rightarrow \text{Obj}(\mathcal{C})$
- $\text{Obj}(\mathcal{C})$ -indekseeritud kujutiste perest $\eta_A : A \rightarrow TA$
- operatsioonist $-^*$, mis igale morfismile $f : A \rightarrow TB$ seab vastavusse morfismi $f^* : TA \rightarrow TB$
 - nn. monaadi **ekstensioon**

selliselt, et iga $f : A \rightarrow TB$ ja $g : B \rightarrow TC$ korral kehtivad järgmised võrdused:

$$\begin{aligned}\eta_A^* &= id_{TA} \\ f^* \cdot \eta_A &= f \\ (g^* \cdot f)^* &= g^* \cdot f^*\end{aligned}$$

Monaadid

Lemma

Monaadid ja Kleisli kolmikud on ekvivalentsed mõisted.

Tõestus

Olgu $f : A \rightarrow TB$ ja $g : A \rightarrow B$

- $(T, \eta, \mu) \implies (T, \eta, -^*)$

$$f^* = \mu_B \cdot Tf$$

- $(T, \eta, -^*) \implies (T, \eta, \mu)$

$$Tg = (\eta_B \cdot g)^*$$

$$\mu_A = id_{TA}^*$$

Monaadid Haskellis

Monaadide klass

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
  (>>)   :: m a → m b → m b
  fail   :: String → m a
  m >> k = m >>= λ_ → k
  fail s  = error s
```


Monaadid Haskellis

Monaadide seadused

$$\begin{aligned} \text{return } x &\gg= f &&= f \ x \\ m &\gg= \text{return} &&= m \\ (m &\gg= \lambda x \rightarrow k) &\gg= f &= m \gg= \lambda x \rightarrow (k \gg= f) \end{aligned}$$

NB!

- Viimases võrduses on eeldatud, et muutuja x ei sisaldu vabalt avaldises f .
- Iga monaad peaks antud võrdusi rahuldama, aga Haskellis kompilaator võrduste kehtivust ei kontrolli. See on programmeerija ülesanne, et tema defineeritud monaad neid võrdusi tõepoolest rahuldab!

Monaadid Haskellis

do-süntaks

$$\begin{array}{lcl} \text{expr} & = & \text{do } \{ \text{stmt}; \dots; \text{stmt} \} \\ \text{stmt} & = & \text{pat} \leftarrow \text{expr} \\ & | & \text{expr} \\ & | & \text{let } \text{decls} \end{array}$$

Transleerimisreeglid

$$\begin{array}{lcl} \text{do } \{ e \} & = & e \\ \text{do } \{ e; \text{stmts} \} & = & e \gg \text{do } \{ \text{stmts} \} \\ \text{do } \{ v \leftarrow e; \text{stmts} \} & = & e \gg= \lambda v \rightarrow \text{do } \{ \text{stmts} \} \\ \text{do } \{ \text{let } \text{decls}; \text{stmts} \} & = & \text{let } \text{decls} \text{ in } \text{do } \{ \text{stmts} \} \end{array}$$

Monaadid Haskellis

Identsusmonaad

```
newtype Id a = Id a
instance Monad Id where
    (Id x) >>= k = k x
    return x     = Id x
```

NB!

Identsusmonaad esitab "puhtaid" (so. ilma efektideta) arvutusi.

Monaadid Haskellis

Maybe monaad

instance *Monad Maybe* where

Nothing $\gg= k = \text{Nothing}$

(Just x) $\gg= k = k\ x$

return x $= \text{Just } x$

fail s $= \text{Nothing}$

Monaadid Haskellis

Olekuteisendusmonaad

```
newtype S s a = S (s → (a, s))  
instance Monad (S s) where  
    S m >>= k = S $ λs →  
        case m s of  
            (x, s') → case k x of  
                S g → g s'  
    return x = S (λs → (x, s))
```

Monaadid Haskellis

Monaadispetsiifilised operatsioonid

- Monaadi baasoperatsioonid *return* ja ($\gg=$) võimaldavad tekitada puhtaid arvutusi ning arvutusi järjestikku komponeerida.
- Et monaad omaks mõtet, peab ta lisaks veel omama lisaoperatsioone, mis on spetsiifilised monaadi poolt esitatavatele efektidele.
- Reeglina on kahesuguseid monaadispetsiifilisi operatsioone:
 - operatsioonid konkreetse efekti tekitamiseks;
 - operatsioonid arvutustulemuse väliseks inspekteerimiseks.

Monaadid Haskellis

Maybe monaadi spetsiifilised operatsioonid

raise :: *Maybe a*

raise = *Nothing*

handle :: *Maybe a* → *Maybe a* → *Maybe a*

handle m h = **case** *m* **of**

Nothing → *h*

Just _ → *m*

Monaadid Haskellis

Olekuteisendusmonaadi spetsiifilised operatsioonid

getS :: *S* *s* *s*

getS = *S* ($\lambda s \rightarrow (s, s)$)

setS :: *s* $\rightarrow$ *S* *s* ()

setS *x* = *S* ($\lambda s \rightarrow ((), x)$)

runS :: *S* *s* *a* $\rightarrow$ *s* $\rightarrow$ (*a*, *s*)

runS (*S* *f*) *s* = *f* *s*

Monaadid Haskellis

Listide monaad

```
instance Monad [] where
    xs >>= f  = concat (map f xs)
    return x  = [x]
    fail s    = []

deadlock :: [a]
deadlock = []

choice :: [a] → [a] → [a]
xs 'choice' ys = xs ++ ys
```

NB!

Listide monaad vastab mittedetermineeritud arvutustele.

Monaadid Haskellis

Keskkondade monaad

```
instance Monad (( $\rightarrow$ ) e) where  
    m >>= f  =  $\lambda e \rightarrow f (m\ e)\ e$   
    return x =  $\lambda e \rightarrow x$   
  
getEnv  :: (( $\rightarrow$ ) e) e  
getEnv  =  $\lambda e \rightarrow e$   
  
withEnv :: (( $\rightarrow$ ) e) a  $\rightarrow e \rightarrow a$   
withEnv f e = f e
```

NB!

(>>=), *return*, *getEnv* vs. *S*, *K*, *I* kombinaatorid.

Monaadid Haskellis

Klass *MonadPlus*

```
class Monad m => MonadPlus m where  
    mzero :: m a  
    mplus :: m a -> m a -> m a
```

Nõutud võrdused

```
mzero 'mplus' m = m  
m 'mplus' mzero = m  
(m1 'mplus' m2) 'mplus' m3  
    = m1 'mplus' (m2 'mplus' m3)  
mzero >>= f      = mzero  
m >> mzero       = mzero
```

Monaadid Haskellis

Maybe monaad

```
instance MonadPlus Maybe where
    mzero           = Nothing
    Nothing 'mplus' m2 = m2
    m1           'mplus' m2 = m1
```

Listide monaad

```
instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

Monaadid Haskellis

Parserite monaad

```
newtype Parser a = P (String → [(a, String)])

instance Monad Parser where
    return a    = P $ λcs → [(a, cs)]
    p >>= f     = P $ λcs → concat
                        $ [ runP (f a) cs' | (a, cs') ← runP p cs ]

instance MonadPlus Parser where
    mzero       = P $ λcs → []
    mplus p q = P $ λcs → runP p cs ++ runP q cs

item  :: Parser Char
item  = P $ λcs → [(head cs, tail cs) | ¬ (null cs)]

runP :: Parser a → String → [(a, String)]
runP (P p) = p
```