Дәріс №8-9

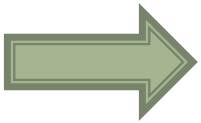
Енгізудің және шығарудың құрылымдары

Модулдер

Haskell программалау тілінде модулдердің жинағынан тұрады. Модулдер екі негізде жұмыс істейді - басқармаға аттың аяларының және деректердің абстракт үлгісінің жаралығынан тұрады.

Бас әріптен басталатын модулдердің аттары болады; Hugs интерпритаторының мәтінінің модулі басқа файлда болады және модулдің аты атпен сәйкес келуі керек. Бұл файл . hs. кеңейтілімде болуы керек.

Практикалық негізде модульдер module кілттік сөзінен басталатын ең үшкен хабарламадан тұрады. Tree модулінің атынан мысал кертірейік.



```
module Tree  ( Tree(Leaf,Branch),  leafList)  where
data Tree a = Leaf a  I  Branch  (Tree a)  (Tree a)
leafList  (Leaf x)  [x]
leafList  (Branch left right)  =  leafList  left ++
leafList right
```

Module кілттік сөзінен кейін модул аты экспортталады. Модулден экспортталатын аттар кілттік сөзден кейін жақшада көрсетіледі. Егер ерекшеленген аттар көрсетілмеген болса, онда барлық аттар модулден экспортталады. Tree (Leaf, Branch) конструкциясы сияқты барлық аттар типі және оның конструктрлары топталған болуы керек. Қысқарту негізінде Tree (. .) жазуын қолдануға болады. Сонымен қатар, берілген мәліметтердің жарты конструкторларын ғана экспорттауға болады.

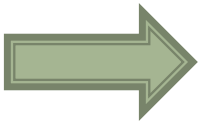
Tree модулін енді кез-келген модулге импорттауға болады.

```
module Main where
import Tree  (Tree(Leaf,Branch),  leafList)
```

Мұнда біз импортталатын түйінділердің тізімдерін көрсеттік, егер оны жіберетін болсақ, модулден экспортталатын барлық түйіндер импортталады.

Егер екі импортталатын модулде әртүрлі түйіндер бір атаумен болса онда келеусіздіктер болады. Ондай қателікті жою үшін qualified кілттік сөзі қолданылады, ол импортталатын модулді анықтайды, объект аттарына түр негізделеді: «Модуль. Объект». Мысалы, Tree модулі үшін:

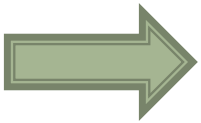
```
module Main where import qualified Tree
leafList = Tree.leafList
```



Абстрактілі мәліметтер типі

Модулдерді қолдану абстрактілі мәліметтерді анықтауға көмектеседі, типтер, ішкі құрылымдар қолданушыдан жасырын болады. Мысалы, берілген сөздің мағынасын қайтаратын қарапайым сөздікті қарастырамыз:

```
module Dictionary where
data Dictionary = Dictionary [(String,String)]
getMeaning :: Dictionary -> String   -> Maybe String
getMeaning [] _ = Nothing
getMeaning ((word,meaning):xs) w     |    w == word
= Just meaning
                                     |   otherwise =
Nothing
```



getMeaning функциясы берілген сөздік және сөз бойынша табылған мағынаны қайтарады (Maybe типін қолдану арқылы). Сөздіктің өзі жұпты сөздікті көрсетеді.

Сөздікті қалай жасауға болады? Модулді қолданушы addWord функциясын анықтауы мүмкін, «Сөз-мағына» деген жұпты қосады және модифицирланған сөздікті қайтарады:

```
import Dictionary
addWord (Dictionary dict) word meaning = Dictionary ((word,meaning):dict)
```

Мұнда қолданушы сөздіктің көрсетілімін көреді және оны қолдануына болады. Болашақта біз сөздіктің берілуін өзгертуіміз де мүмкін. Тізім – іздеу жүйесі үшін қолданудың оңай тәсілі емес. Одан да хеш – кестесін және іздеу ағашын пайдаланған өте тиімді.

Егер Dictionary типі ашық болса, қолданушының пайдаланатын программасынсыз біз оны бұза алмаймыз. Ішкі модулдің көрсетілімін пайдаланушыдан жасыру үшін, Dictionary типін абстрактілі ету керек. addWord функциясын және бос сөздікті беретін модулде emptyDict мағынасын анықтаймыз. Пайдаланушы Dictionary типінің мағынасымен кеңейтілген функция көмегімен ғана қарым-қатынас жасай алады:

```
module Dictionary (Dictionary, getMeaning, addWord, emptyDict) where data Dictionary = Dictionary [(String,String)]
getMeaning :: Dictionary -> String   -> Maybe String
getMeaning [] _ = Nothing
```

```
getMeaning ((word, meaning)   :   xs) w   |   w == word = Just meaning
                                          |  otherwise   = Nothing
addWord (Dictionary dict) word meaning = Dictionary ((word,meaning):dict)
emptyDict = Dictionary []
```

Енгізу-шығару операциясы

Енгізу-шығару жүйесі Haskell тілінде толық анықталған, бірақ енгізу-шығару жүйесінде императитвті тілдерде өз мүмкіндіктерін жібермейді. Императивті тілдерде программа жалғастырушы әрекеттерді көрсетеді,онда қоршаған ортаның мазмұнын ескереді және өзгертеді.

Haskell тілінде енгізу-шығару жүйесі монад концепция-сының манында құрылған. Бірақ программалауда енгізу-шығару монад көптеген түсініктерді қажет етпейді, басқа қарапайым арифметикалық амалдарды алгебрада шешу

жүйелері сияқты. Сондықтан біз, енгізу-шығару жүйесін монадқа жүйелемей, Дәрісларда қарастырылады. Haskell тілімен әрекет орындалмайды, тек анықталады. Әрекеттің анықталуы, оның орындалғанын көрсетпейді.

Енгізу-шығару операциясының базалық негізі

Әрбір әрекет мағынаны қайтарады. Басқадан мағынасын қайтаратын типтер жүйесі ШО типімен байланысты. Мысалы, getChar:функциясын қарастырайық:

```
getChar   ::   IO Char
```

IO Char көрсетеді, getChar шақыруда әр түрлі әрекеттер орындайды, символды қайтаратын. Нәтижені қайтармайтын әрекет, IO () типін қолданады. О символы бос типті көрсетеді (тілде

void типіне ұқсас). Мысалы, putChar: функциясы:

```
putChar :: Char -> IO ()
```

Ол символды қабылдап ешқандай қызықты нәрсені қайтармайды. Әрекеттер бір-бірімен >>= операторлары арқылы байланысады. Бірақ біз do-нотацияны пайданатын боламыз. Do кілттік сөзі қатар-қатармен орындалатын операторлары жалғасын табады. Оператор әр түрлі әрекет болуы мүмкін, <- әрекетімен байланыстыратын әрекет, үлгі де болуы мүмкін. do-нотациясында ерекшелеудің сол ережесі болады, let және where кілттік сөздері сияқты. Символды санайтын және баспаға беретін қарапайым программа.

```
main :: IO ()
```

```
main = do c <- getChar
          putChar  c
```

main атын қолдану мұнда жай емес, main функциясы Main модулі сияқты Haskell тілінк кірудің негізгі нүктесі болып табылады, Си-дағы main функциясы сияқты.

Оның типі IO () сәйкес болуы тиіс. Көрсетілген программа екі әрекетті бір мезетте орындайды: символдарды санайты, әрекеттің мағынасын негіздеді. Мысалы, сізге 'y' тең болатын, символды санайтын және True қайтаратын, ready функциясын анықтау керек.

```
ready   ::   IO Bool
ready = do c <-    getChar
c == 'y' -- Ошибка!!!
```

Программа жұмыс істемейді, өйткені екінші do операторы әрекет емес, булдік мағына болып табылады. Бізге булдік мағынаны нәтиже негізінде қайтаратын, булдік мағына алу керек және әрекет құру керек. Ол үшін return функциясы қолданылады:

```
return  ::   a -> IO a
```

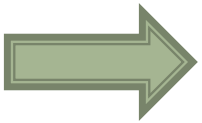
return функциясы әрекеттің жалғасуын аяқтайды. Сонымен, ready мына жүйемен анықталады:

```
ready   ::   IO Bool
ready = do c <-  getChar
    return (c == 'y')
```

Біз енді енгізу-шығаруда күрделі функцияларды анықтайымызға болады. getLine функциясы жолды қайтаратын, жолдар символының соңын клавиатурамен саналатын аяқтауда қолданады.

```
getLine   ::   IO String
getLine = do c <- getChar
if c ==  '\n'
then return ""
else do l <- getLine
return  (c:l)
```

return функциясы ензігу-шығару әрекетінің аумағында қарапайым әрекетті енгізеді. Кері әрекетті қалай көреміз? Енізу-шығару әрекетін қарапайым негізде орындауға бола ма? Әрине, Жоқ. f : : Int -> Int функциясы енгізу-шығару операциясын орындай алмайды, өйткені IO қайтарылатын мағынада типтер анықталмайды.



Енгізу-шығару стандартты операциялары

Келесі әрекеттерді және енгізу-шығару файлдарымен жұмыс істеуде әрекеттерді және типтерді анықтауды қарастырамыз (олар IO моулінде анықталған).

type FilePath = String - - имена файлов в файловой системе

openFile :: FilePath -> IOMode -> IO Handle

hClose :: Handle -> IO ()

data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode

Файлды ашу үшін openFile функцияны қолданылады, онда файл аты және режимі беріледі. Сонымен қатар дескриптор файлы (типа Handle) құрылады, сонынан міндетті түрде hClose функциясының көмегімен жабу керек.

Файлдан символдарды және жолдарды санау үшін келесі функциялар қолданылады:

hGetChar :: Handle -> IO Char

hGetLine :: Handle -> IO String

Файлға жазу үшін келесі функция қолданылады:

hPutChar :: Handle -> Char -> IO O hPutStr :: Handle -> String -> IO ()

Клавиатурадан санау және экранға енгізу үшін келесі функциялар қолданылады:

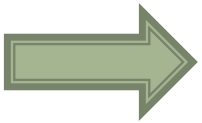
getChar : : IO Char

getLine : : IO String

readLn : : IO Int - - чтение числа целого типа

putChar : : Char -> IO ()

putStr : : String -> IO ()



Төменде көрсетілген мысалда, тізімнің элементтерінің суммасы шығарылады:

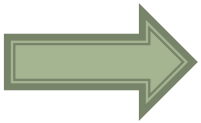
```
main :: IO ()
main = do
        putStr "Enter a list of numbers: "
        listStr <- getLine
        print (sum (read listStr))
```

Результат выполнения:

```
Main> main
Enter a list of numbers: [1,2,3]
6
()   ::   IO ()
```

Сонымен қатар ,келесі функция өте қажетті:

```
hGetContents   ::  Handle  -> IO String
```



Ол барлық файлды үлкен бір жол сияқты санайды. Бір қрағанда бұл функция өте тиімсіз, бірақ, қойылған есептеулерде файлдан қанша символ болса сонша символ саналыд, бірақ одан көп емес.

Мысалы

Файлды кшіру программасын жазамыз. Ол клавиатурадан екі файлды санайды, шығатын және бағытталған, және басқа файлды бір файлға көшіреді.

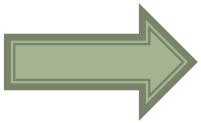
- Функция шақыру билетін шығарады, сонымен қатар файл атын санайды.

- оны көрсетілген режимде ашады

```
getAndOpenFile prompt mode = do putStr prompt
      name    <-    getLine
     openFile name mode
```

```
main = do fromHandle    <-    getAndOpenFile "Copy from:
" ReadMode
    toHandle   <-   getAndOpenFile "Copy to" WriteMode
    contents <- hGetContents fromHandle
    hPutStr toHandle contents
    hClose toHandle
    putStr "Done."
```

hGetContents функциясын қолданғандығымызға қарамастан, барлық мазмұндағы файлдар жадыда болмайды, оған шмасы келгенше оны оқып дискіге жазады. Ол компьютердің оперативті жадысынан сәйкес келетін, одан да көп файлды жазуға мүмкіндік береді. Ағымдағы файл жбылады, егер одан сиңғы символ оқылатын болса. Программаның жолдық командасының параметрлеріне қатынас жасау үшін



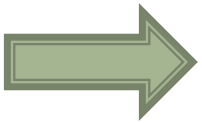
System модулінде анықталған келесі функцияларды қолдануға болады:

```
getArgs :: IO [String]
```

Бұл функция жолдар тізімін қайтарады, командалық жолдар парамері болып табылатын

Си программасында argv массивіне сәйкес келетін. Көшіру программасын мына жолмен анықтауға болады:

```
main = do args  <-   getArgs
    copyFile
    putStr  "Done.“
copyFile   [from,  to]   = do fromHandle   <-   openFile
from ReadMode
        toHandle   <-   openFile to WriteMode
```



```
contents <- hGetContents fromHandle
        hPutStr toHandle  contents
        hClose toHandle
copyFile _ = error "Usage:   copy <from> <to>“
```

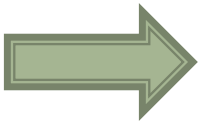
Бұл программа шығатын және бағытталған файлдарды командалық жолдан қабылдайды.

copyFile функциясы қателер туралы хабарлама шығарады, егер программаға аргументтер саны қате берілген болса.

Қолданылатын программалырды құру

Интерпретаторларды қолдану арқылы біз Haskell программасын әлі пайдаланудамыз.

Бірақ жеке қолданылатын программа құру үшін де мүмкіндіктер бар, кейбір программаларды орындау үшін интерпретаторды пайдаланбаса да болады.



Ол үшін ghc командасы арқылы шақырылатын, Glasgow Haskell Compiler компиляторы қолданылады.

Компиляциялау үшін жолдық командаға келесі әрекеттерді енгізу керек:

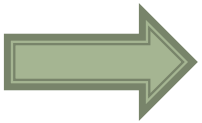
ghc -o Main.exe Main.hs

Программада қате кездесетін болса, экранға қателер туралы хабарлама шығады. Егер қате болмаған жағдайда, компилятор орындауға жіберуге болатын қолданатын файл құрады. Программа орындаудың нәтижесі:

--Main.hs

main :: IO () main = do

print("Enter first number:")



```
numlStr <- getLine
print("Enter second number: ")
num2 <- readLn
print("Sum=")
print( read numlStr+num2)
```

будет:

```
C:\ghc\ghc-6.2\bin>Main.exe
"Enter first number:"
4
"Enter second number:   "
3
"Sum=“
7
```