**Ввод и вывод**

Функции ввода/вывода — функции с побочными эффектами.

В Haskell такие функции запрещены.

Решение: выделение операций ввода/вывода в отдельный «подъязык», в рамках которого функции с определённым типом могли выполнять действия с побочными эффектами (изменения внешнего окружения — устройств вывода) и быть недетерминированными.

Однако вне этого подъязыка язык Haskell остаётся чистым.

В языке Haskell используется понятие «действие» для описания специальных функций, которые должны иметь и специальный тип.

Например, функции из Prelude

putStrLn – вывести строку, заканчивающуюся переводом строки:

putStrLn :: String -> IO ()

getChar, которая считывает с клавиатуры один символ:

getChar :: IO Char

Каждое действие ввода/вывода должно возвращать какое-то значение. Для того чтобы различать эти значения от базовых, типы этих значений как бы обёрнуты контейнерным типом IO. Поэтому любое действие ввода/вывода будет иметь в сигнатуре своего типа символы IO.

Действия в отличие от обычных функций выполняются, а не вычисляются, для этого необходимо использовать ключевое слово do либо специальные методы, определённые в классе Monad.

Но существует одно действие, которое выполняется само. Это функция main, которая является точкой входа в откомпилированные программы. Именно поэтому тип функции main должен быть IO () — это действие, которое автоматически выполняется первым при запуске программы.

Тип IO () — это тип действия, которое ничего не возвращает в результате своей работы. Иные действия, имеющие некоторый результат, который можно получить в программе, должны возвращать другой тип. Так, к примеру, действие функции getChar имеет тип IO Char.

Любые действия связываются в последовательности при помощи ключевого слова do. При помощи него можно связывать вызовы функций (в том числе и использовать операторы ветвления if и case), получение значений в образцы (при помощи символа (<-)) и множество определений локальных переменных (ключевое слово let).

Пример 1. Программа читает символ с клавиатуры и тут же выводит:

main :: IO ()

main = do

c <- getChar

putChar c

Пример 2. Функция isReady возвращает True, если нажата клавиша «y», и значение False в остальных случаях:

isReady :: IO Bool

isReady = do

c <- getChar

returen (c == ’y’) -- :: IO Bool

Метод return из простого типа данных делает контейнерный, в котором хранится значение исходного типа.

Пример 3. Функция считывает строку символов с клавиатуры:

getString :: IO String

getString = do

c <- getChar

**if** (c == ’\n’)

**then** return "" -- можно без do

**else** do

cs <- getString

**return** (c:cs)

В частях then и else условного выражения и выражениях для альтернатив в операторе case необходимо также использовать последовательность действий, оформленную в виде списка do, если действий больше одного.

Действия ввода/вывода являются обычными значениями, которые можно использовать там, где можно использовать обычные данные

Пример 4. Выполнение списка действий:

todoList :: [IO ()] – список действий

todoList = [putChar ’a’,do {putChar ’b’; putChar ’c’},

do {c <- getChar; putChar c]

-- функция для выполнения списка действий

sequence :: [IO ()] -> IO ()

sequence [] = return ()

sequence (a:as) = do {a; sequence as}

Функция putString выводит заданную строку

:

putString :: String -> IO ()

putString s = sequence (map putChar s)

**Обработка исключений**

Для обработки ошибочных ситуаций при вводе/выводе используется такой же механизм, как и в императивных языках программирования (С++). В Haskell нет специального синтаксиса, используются функции.

Ошибка ввода/вывода имеет тип IOError, а обработчик исключительной ситуации обязан иметь тип IOError -> IO a. Для связывания обработчика с кодом, в котором возможно возникновение ошибочной ситуации, имеется специальная функция catch, имеющая тип

catch :: IO a -> (IOError -> IO a) -> IO a

Первый аргумент – действие или список действий, оформленный при помощи ключевого слова do. Второй – обработчик исключений.

Если действие выполнено успешно, возвращается результат этого действия. Если возникла ошибка, то она передается обработчику исключений в качестве операнда типа IOError, после чего выполняется сам обработчик. Результат его работы возвращается в качестве окончательного результата.

Пример 5. Вложенные обработчики ошибок

-- обрабатывается обнаружение символа конца файла

getChar’ :: IO Char

getChar’ = catch getChar eofHandler

**where**

eofHandler e = **if** (isEofError e)

**then** return ’\n’

**else** ioError e

-- ioError обрабатывает другие ошибки

getString’ :: IO String

getString’ = catch getString’’ (\e -> return ("Error: " ++ show e))

**where**

getString’’ = do

c <- getChar’

**if** (c == ’\n’)

**then** return ""

**else** do

cs <- getString’

return (c:cs)

На примере определения этих функций видно, что можно использовать вложенные друг в друга обработчики ошибок. В функции getChar’ отлавливается ошибка, которая возникает при обнаружении символа конца файла. Если ошибка другая, то при помощи функции ioError она отправляется дальше и ловится обработчиком, который «сидит» в функции getString’.

Для определённости в языке Haskell предусмотрен обработчик исключений по умолчанию, который находится на самом верхнем уровне вложенности. Если ошибка не поймана ни одним обработчиком, который написан в программе, то её ловит обработчик по умолчанию, который выводит на экран сообщение об ошибке и останавливает программу.

Пример 5. Копирование файла

import IO

copyfile f1 f2 = do {

fin <- catch (openFile f1 ReadMode) (\\_ -> error "Cannot open f1" )

; fto <- catch (openFile f2 WriteMode) (\\_ -> error "Cannot open f2" )

; ss <- hGetContents fin

; hPutStr fto ss

; hClose fto

; putStr "Done."

}

Пример 5. Копирование файла без обработчиков

copyf f0 f1 = do {

ss <- readFile f0

; writeFile f1 ss

}