Haskell

Ввод-вывод
Параллелизм

Выполнили:
Костюхин Алексей
Тасаков Антон
Теплов Андрей
Студенты гр. 5030102/10201

Монада IO

```haskell
main :: IO ()
main = do
    putStrLn "Введите ваше имя:"
    name <- getLine
    putStrLn ("Привет, " ++ name ++ "!")
```

Монада **IO** в Haskell используется для работы с вводом-выводом, включая взаимодействие с пользователем, чтение и запись в файлы, сеть и другие эффекты реального мира. Она является фундаментальной частью Haskell, позволяя чистому функциональному языку безопасно работать с побочными эффектами

Работа с консолью

Функция	Тип	Описание
`getLine`	`IO String`	Считывает строку текста (до нажатия Enter) из стандартного ввода.
`getChar`	`IO Char`	Считывает один символ из стандартного ввода.
`readLn`	`Read a => IO a`	Считывает строку, парсит её в значение типа `a` с использованием `Read`.
`putStrLn`	`String -> IO ()`	Выводит строку в стандартный вывод и добавляет символ перевода строки.
`putStr`	`String -> IO ()`	Выводит строку в стандартный вывод без добавления символа перевода строки.
`print`	`Show a => a -> IO ()`	Выводит значение любого типа с использованием `show`, добавляет символ перевода строки.

Работа с консолью. Пример

```haskell
main :: IO ()
main = do
    putStrLn "Введите ваше имя:"
    name <- getLine
    putStrLn ("Привет, " ++ name ++ "!")

    putStrLn "Введите ваш возраст:"
    age <- readLn :: IO Int
    putStr "Ваш возраст: "
    print age

    putStrLn "Нажмите любую клавишу..."
    _ <- getChar
    putStrLn "Завершение..."
```

Работа с файлами

Функция	Тип	Описание
`readFile`	`FilePath -> IO String`	Считывает содержимое файла целиком в виде строки.
`writeFile`	`FilePath -> String -> IO ()`	Записывает строку в файл, перезаписывая его содержимое.
`appendFile`	`FilePath -> String -> IO ()`	Добавляет строку в конец файла без удаления его текущего содержимого.

Функция	Тип	Описание
`openFile`	`FilePath -> IOMode -> IO Handle`	Открывает файл с указанным режимом (`ReadMode`, `WriteMode`, `AppendMode`, `ReadWriteMode`) и возвращает дескриптор (`Handle`).
`hClose`	`Handle -> IO ()`	Закрывает ранее открытый файл, освобождая ресурсы.
`hGetContents`	`Handle -> IO String`	Считывает содержимое открытого файла в виде строки.
`hPutStr`	`Handle -> String -> IO ()`	Записывает строку в открытый файл через дескриптор (`Handle`).
`hGetLine`	`Handle -> IO String`	Считывает одну строку из открытого файла через дескриптор (`Handle`).
`hIsEOF`	`Handle -> IO Bool`	Проверяет, достигнут ли конец файла для указанного дескриптора (`Handle`).

Работа с файлами. Пример

```haskell
main :: IO ()
main = do
    writeFile "example.txt" "Привет, Haskell!\n"
    appendFile "example.txt" "Это добавленная строчка.\n"

    content <- readFile "example.txt"

    putStrLn "Содержимое файла:"
    putStrLn content
```

```haskell
main :: IO ()
main = do
    handle <- openFile "example.txt" WriteMode
    hPutStr handle "Привет, Haskell!\n"
    hPutStr handle "Это работа через дескриптор.\n"
    hClose handle
```

Конкурентность и параллелизм

- **Конкурентность** — это когда несколько задач исполняются одновременно, но не обязательно параллельно (на одном процессоре, с переключением контекста)
- **Параллелизм** — это когда вычисления действительно выполняются одновременно (на нескольких ядрах процессора)

Модуль Control.Parallel

Функция	Тип	Описание
`par`	`a -> b -> b`	Запускает вычисление первого аргумента (`a`) параллельно с вычислением второго (`b`). Результат — второй аргумент (`b`).
`pseq`	`a -> b -> b`	Гарантирует, что первый аргумент (`a`) будет вычислен до начала вычисления второго (`b`). Результат — второй аргумент.

```haskell
import Control.Parallel (par, pseq)

parallelExample :: Integer -> Integer -> Integer
parallelExample x y =
    let a = expensiveComputation x
        b = expensiveComputation y
    in a `par` (b `pseq` (a + b))

expensiveComputation :: Integer -> Integer
expensiveComputation n = sum [1..n]

main :: IO ()
main = print (parallelExample 1000000 1000000)
```

Модуль Control.Parallel.Strategy

Функция	Тип	Описание
`using`	`a -> Strategy a -> a`	Применяет стратегию параллельного вычисления (`Strategy`) к значению, возвращая его после вычисления.
`rpar`	`Strategy a`	Параллельно запускает вычисление значения.
`rseq`	`Strategy a`	Гарантирует строгую последовательность: значение полностью вычисляется перед продолжением программы.
`parList`	`Strategy a -> Strategy [a]`	Применяет стратегию параллельного вычисления ко всем элементам списка.
`parBuffer`	`Int -> Strategy a -> Strategy [a]`	Параллельно вычисляет элементы списка с буферизацией (заранее вычисляет указанное количество элементов).

Пример

```haskell
import Control.Parallel.Strategies (using, rpar, parList)

parallelSum :: [Int] -> Int
parallelSum xs = sum xs `using` parList rpar

main :: IO ()
main = print (parallelSum [1..1000000])
```

Модуль Control.Concurent

Функция/Тип	Тип	Описание
`MVar`	`MVar a`	Многопоточная переменная, обеспечивающая синхронизацию между потоками. Поддерживает операции записи и чтения.
`newEmptyMVar`	`IO (MVar a)`	Создаёт новую пустую `MVar`, готовую для записи данных.
`putMVar`	`MVar a -> a -> IO ()`	Записывает значение в `MVar`. Если переменная уже занята, поток блокируется до её освобождения.
`takeMVar`	`MVar a -> IO a`	Извлекает значение из `MVar`. Если переменная пуста, поток блокируется до появления значения.

Функция	Тип	Описание
`forkIO`	`IO () -> IO ThreadId`	Создаёт новый поток (`Thread`) для выполнения указанного действия `IO`. Возвращает идентификатор потока.
`yield`	`IO ()`	Освобождает процессор, позволяя другим потокам выполниться.
`threadDelay`	`Int -> IO ()`	Приостанавливает выполнение текущего потока на указанное количество микросекунд.
`threadKill`	`ThreadId -> IO ()`	Завершает выполнение потока с указанным `ThreadId`.
`threadId`	`ThreadId`	Уникальный идентификатор потока. Используется для управления потоками, созданными с помощью `forkIO`.

Пример

```haskell
import Control.Concurrent

main :: IO ()
main = do
    -- Создаём пустую MVar
    mvar <- newEmptyMVar

    -- Создаём поток, который будет записывать данные в MVar
    forkIO $ do
        putStrLn "forkIO: Записываем данные в MVar..."
        threadDelay 1000000  -- Задержка 1 секунда
        putMVar mvar "Привет из forkIO!"  -- Записываем строку в MVar
        putStrLn "forkIO: Данные записаны."

    -- Главный поток извлекает данные из MVar
    putStrLn "main: Ждём данные из MVar..."
    message <- takeMVar mvar              -- Извлекаем строку из MVar
    putStrLn $ "main: Полученное сообщение: " ++ message
```

Модуль Control.Concurent.STM

Функция/Тип	Тип	Описание
`TVar`	`TVar a`	Многопоточная переменная, используемая для атомарных транзакционных операций в рамках STM (Software Transactional Memory).
`newTVar`	`a -> STM (TVar a)`	Создаёт новую транзакционную переменную (`TVar`) с начальным значением.
`readTVar`	`TVar a -> STM a`	Считывает значение из `TVar` внутри транзакции.
`writeTVar`	`TVar a -> a -> STM ()`	Записывает новое значение в `TVar` внутри транзакции.
`atomically`	`STM a -> IO a`	Выполняет транзакцию STM атомарно, преобразуя её в действие `IO`.

Пример

```haskell
import Control.Concurrent.STM

main :: IO ()
main = do
    counter <- atomically $ newTVar 0
    atomically $ do
        value <- readTVar counter
        writeTVar counter (value + 1)
    finalValue <- atomically $ readTVar counter
    putStrLn $ "Значение счётчика: " ++ show finalValue
```