

The do notation

In order to define *interactive* Haskell programs, we need a way for input and output actions to *alternate*—i.e., we need to define IO actions as sequences of smaller IO actions.

The **do** notation provides a convenient way

- to combine I/O actions in sequence, to make compound actions
- to give names to values delivered by I/O actions, in order to refer to them in subsequent actions

do expressions can look remarkably like imperative programs!

An alternative definition of `putStrLn` using **do**:

```
> putStrLn :: String -> IO ()
> putStrLn s = do putStrLn s
>               putChar '\n' --defines do's type
```

A definition of `putStr` using **do**:

```
> putStr :: String -> IO ()
> putStr [] = return ()
> putStr (c:cs) = do putChar c; putStr cs
                    --using {;} instead of layout
```

A function that returns an action which outputs a given string a given number of times:

```
> putNtimes :: Int -> String -> IO ()
> putNtimes n s
>   = if n < 1
>     then return ()
>     else do putStrLn s
>             putNtimes (n-1) s
```

Naming input values

Within a **do** expression, values *delivered* by I/O actions can be named. The names can then be used in *subsequent* expressions.

Example: an action which reads one line and outputs its reverse:

```
> rev1line :: IO ()
> rev1line = do line <- getLine
>              putStr (reverse line)
```

The “<-” has much the same meaning as in the list comprehension:

The construct

```
pattern <- action
```

introduces a new *pattern* (often just a name)

- whose names' scope is the *rest* of the **do**-construct
- whose value is the value **delivered** by *action*

Note the distinction between <- and = :

- `getLine`
- *the value delivered by* `getLine`

are *not the same* (they're not even the same type).

Hence a definition such as

```
> rev1line = putStr (reverse getLine)
```

would be a **type error**.

Another way to introduce new names into a **do** construct is to use the construction

```
let pattern = expression
```

which introduces a *pattern*

- whose names' scope is the *rest* of the **do**-construct
- whose value is the value of the *expression*

Example:

```
> rev2lines :: IO ()
> rev2lines = do line1 <- getLine
>               line2 <- getLine
>
>               let r1 = reverse line1
>                   r2 = reverse line2
>
>               putStrLn (r2 ++ r1)
>               putStrLn (r1 ++ r2)
```

By the way, **let** can also be used this way in list comprehensions. For example, instead of

```
[ (sqrt y +1, sqrt y -1) | y <- someList]
```

we can write

```
[ (x+1,x-1) | y <- someList, let x = sqrt y ]
```

Here the name *x* enables the expression `sqrt y` to be evaluated only once.

Interactive programs

An action which obtains an integer from the keyboard:

```
> getInt :: IO Int
> getInt =
>   do s <- getLine
>      case reads s of
>        [(n,"")] -> return n
>        _        -> do putStrLn "bad integer!"
>                       getInt
```

An action which reads integers from successive lines until 0 is read, and delivers their sum:

```
> sumInts :: IO Int
> sumInts = do n <- getInt
>             if n == 0 then return 0
>             else do m <- sumInts
>                   return (n+m)
```

A program that outputs a greeting, reads some integers, and prints their sum:

```
> sumInter :: IO ()
> sumInter
>   = do putStrLn "Enter integers, one per line, "
>        putStrLn "ending with 0.\n"
>        sum <- sumInts
>        putStrLn "Sum = "
>        print sum
```

Monads

The type-constructor `IO` is one instance of a class of type constructors called *monads*.

Informally, what monadic types provide is a means of hiding details we don't need—and prefer not—to be concerned with. Monadic types are thus a kind of *abstract data type*.

The `IO` monad hides such details as the state of the input stream, while allowing values to be extracted from it in ways that preserve referential transparency.

What distinguishes monadic types from other abstract data types is that every monadic type `m` provides a means (represented by the `do` construct) of

- defining `m`-values by sequential composition of `m`-values
- introducing values of other types into an `m`-sequence (`return`)
- extracting values of other types from an `m`-sequence (`name <- ...`)

The monadic nature of `IO` actions is what enables the state of the input stream to be passed from one `IO` action to the next without ever being visible in the Haskell code.

Parsing: another practical application of monads.

Parsing is a process of *recognition*, and a parser is a *recognizer*.

- High-level parsers recognize items such as *expressions, terms, and factors*.
- Low-level parsers recognize items such as `'C'`, `'+'`, or individual letters or digits.

In Haskell, high-level parsers are built from low-level parsers using a toolkit of *parser-combining* functions, or *combinators*.

A parser examines the *leading edge* of a string and either *succeeds* or *fails*.

- If it succeeds, it *delivers* a value derived from the item it has recognized (sometimes this is the item itself).
- If it fails, it delivers nothing.

(In this implementation of parsers, what they *deliver* is returned in a list; delivering "nothing" means returning an empty list. Defining parsers as monads hides these details, and enables us to focus on *what* is delivered, rather than on *how* it is delivered.)

The nature of the value delivered by a parser when it succeeds depends on the purpose for which the parser is designed. Examples:

- an *expression* parser could deliver the expression's value, or its postfix representation
- a *numeral* parser could deliver the numeral's value
- a *name* parser could deliver the name's spelling
- a *comment* parser could deliver `()`

The type-constructor `Parser` is a monad defined so that

`Parser a`

is a *parsing action* which, applied to a `String`, attempts to recognize something at the `String`'s leading edge. The type variable `a` is the type of the values the parser delivers when it succeeds.

(Unlike `IO`, `Parser` is defined as a monad purely for convenience; here there's no risk that referential transparency could be compromised.)

Two elementary parsers:

`item :: Parser Char`

When `item` is used to parse a string (`c: _`) it delivers `c`; `item` fails only when it is applied to `[]`.

`mzero :: Parser a`

When `mzero` is used to parse any `String`, it fails.

Two parser-generating functions:

`return :: a -> Parser a`

When `(return x)` is used to parse a string, it delivers `x`.

`sat :: (Char -> Bool) -> Parser Char`

When `(sat p)` is used to parse a string (`c: _`), it delivers `c` if `p c ==> True`; otherwise, `sat p` fails.

A `Parser`—like an `IO` action—is **not** a function, so it can't be applied to a string directly:

```
ParseLib> item "xyz"
ERROR: Type error in application
*** Expression   : item "xyz"
*** Term        : item
*** Type        : Parser Char
*** Does not match : a -> b
```

Instead, we use the function

```
papply :: Parser a -> String -> [(a,String)]
```

to use parsers on strings.

```
ParseLib> papply item "xyz"
[('x', "yz")]
```

That is,

- a parser `p` is **not** a function.
- `papply p` **is** a function (whose argument type is `String`).

When `papply p s` succeeds, it returns a *pair* containing

- the delivered item (call it *r*) found at the beginning of *s*
- the remainder of *s* after removal of *r*

The pair is returned in a list so that failures can be handled by returning an empty list (signifying the delivery of nothing):

```
ParseLib> papply item ""
[]
```