

An application of `get` (with more uses of `Maybe`)

A floating-point-numeral parser:

```

> float :: Parser Float
> float =
>   do sign <- optional (char '-')
>     int  <- digits
>     fpart <- optional (do char '.'
>                         ds <- digits
>                         return ('.': ds) )
>     let mag = case fpart of
>                 Just frac -> read (int ++ frac)
>                 Nothing  -> read int
>     return (case sign of
>               Nothing -> mag
>               _       -> - mag)

> digits :: Parser String
> digits = do d <- digit
>            ds <- many digit
>            return (d:ds)

> optional :: Parser a -> Parser (Maybe a)
> optional p = do x <- p
>                return (Just x)
>                +++
>                return Nothing

```

A demonstration function:

```

> double :: IO ()
> double = do f <- get "input a number: " float
>            putStrLn ("2 * " ++ show f ++ " = "
>                      ++ show (2*f))

```

A session:

```

Main> double
input a number: 9876,5432
... syntax error
input a number: 9876.5432
2 * 9876.5432 = 19753.0864

```

Maybe and Monad

Because `Maybe` is an instance of `Monad`, functions that return `Maybe` values can be composed using `do`:Example: Define a function `process` so that`process ns j k`finds the `j`th and `k`th elements of the list `ns`, and returns their sum. If either of `j` and `k` is not a valid index of `ns`, `process` should return `0`.

A straightforward solution:

```

> process :: Num a => [a] -> Int -> Int -> a
> process ns j k
>   | 0 <= j && j < lns &&
>     0 <= k && k < lns   = ns!!j + ns!!k
>   | otherwise           = 0
>   where
>     lns = length ns
>     (!! ) :: [b] -> Int -> b
>     (x:_) !! 0      = x
>     (_:xs) !! n | n>0 = xs !! (n-1)
>     (_:_) !! _     = error "negative index"
>     []    !! _     = error "index too large"

```

This function

- requires 3 traversals of `ns`
- fails when `ns` is infinite

Using `Maybe` as a monad:

```

> process :: Num a => [a] -> Int -> Int -> a
> process ns j k = case do m <- ns !!! j
>                      n <- ns !!! k
>                      return (m+n)
>                    of Just s -> s
>                       Nothing -> 0
>   where
>     infixl 9 !!!
>     (!!!) :: [a] -> Int -> Maybe a
>     (x:xs) !!! 0      = Just x
>     (_:xs) !!! n | n>0 = xs !!! (n-1)
>     _             !!! _ = Nothing

```

This function

- requires only 2 traversals of `ns`
- works even when `ns` is infinite

```

MaybeTest> process [1..10] 3 5
10
MaybeTest> process [1..10] 12 5
0
MaybeTest> process [1..10] 3 50
0

```

Conclusion: Like exceptions, `Maybe` provides a way to distinguish between “normal” and “exceptional” processing.

- In C++ and Java, exceptions provide alternative *execution sequences* (`throw` → `catch`).
- In Haskell, `Maybe` provides an alternative *value* (`Nothing`).

The Big Picture: The semester in context

Software's problem: **Unmastered Complexity**

How have languages evolved in response?

- machine-oriented → programmer-oriented

This reflects technological advances:

- Computers are fast, inexpensive and abundant

Moore's Law: Transistor density doubles every 18 months.

—Gordon Moore, 1965

Understanding this phenomenal (exponential) growth:

If cars had improved at the same rate as information technology over the past 35 years, with respect to size, cost, performance, and energy efficiency, then an automobile would

- be the size of a toaster
- cost \$200
- travel 100,000 miles per hour
- get 150,000 miles from one gallon of fuel

During these 35 years, the "hardware" of the human brain hasn't improved at all.

The obvious lessons:

- We should make computers do more so humans can do less.
- We should provide the brain with better "software", i.e., better methods of thinking.

How do we apply this conclusion in programming?

We attack complexity by means of

- programming languages that are more abstract (higher-level, simpler)
- more and better abstractions to use in building software
- better programming-language technology for building and using abstractions

Abstraction in Language Evolution

Machine-oriented languages

- machine languages, assembly languages
- one machine operation per instruction

1st-generation imperative languages (1960)

- *expressions*: many machine operations per statement; readable, modular, and abstract
- *variables*: representing storage locations
- *procedures* (named subprograms with parameters): the first appearance of programmer-defined abstractions

Examples: FORTRAN, ALGOL-60, COBOL

2nd-generation imperative languages (early '70s)

- control structures (**if**, **case**, **for**, **while**) replace the **go-to**
- programmer-defined data types

Examples: C, Pascal

3rd-generation imperative languages (late '70s)

- modules restrict names' visibility: software gets firewalls, watertight compartments
- abstract data types (not fully baked)
- support for defining and coordinating concurrent processes

Examples: Modula-2, Ada

4th-generation imperative languages (mid '80s)

- full support for ADTs (auto-initialization, deep copying)
- ad-hoc polymorphism (overloading)
- parametric polymorphism (templates)

Example: C++

5th-generation (object-oriented) imperative languages ('90s)

- ADTs plus type derivation (extension), dynamic polymorphism
- *garbage collection*—memory management becomes a system responsibility (not C++)

Examples: (C++), Smalltalk, Eiffel, Java, Oberon

Another trend: Notational sprawl (the "kitchen-sink" syndrome)

- languages grow bigger, more complex
- features are added *ad hoc*
- features interact in unexpected ways

Examples: C++, Ada, Fortran-90

Counterexamples: Oberon, Java, Eiffel