# Functions from 'Alternative' type class [duplicate]

**Possible Duplicate:**
Haskell: some and many
Haskell - What is Control.Applicative.Alternative good for?

What are the functions `some` and `many` in the `Alternative` type class useful for? Docs provide a recursive definition which I was unable to comprehend.

haskell functional-programming typeclass

asked Oct 6 '11 at 6:45
missingfaktor
53.2k ●26 ●191 ●308

**marked** as duplicate by Landei, FUZxxl, hammar, Donal Fellows, YOU Oct 8 '11 at 3:25

This question has been asked before and already has an answer. If those answers do not fully address your question, please ask a new question.

@Landei: I read the answer in that thread, and I still don't get it. – missingfaktor Oct 6 '11 at 11:04

I just said this question is a duplicate, not that the original one had a good answer :-) Although it was good enough for me: I figured out that these functions are very likely not interesting for me... – Landei Oct 6 '11 at 11:32

@Landei: I am reaching about the same conclusion as you did. :-) – missingfaktor Oct 6 '11 at 11:43

If you are going to close this question, please merge it with @Landei's. Don't delete it. – missingfaktor Oct 7 '11 at 6:23

## 2 Answers

`some` and `many` can be defined as:

```
some f = (:) <$> f <*> many f
many f = some f <|> pure []
```

Perhaps it helps to see how `some` would be written with monadic `do` syntax:

```
some f = do
  x <- f
  xs <- many f
  return (x:xs)
```

So `some f` runs `f` once, then "many" times, and conses the results. `many f` runs `f` "some" times, or "alternatively" just returns the empty list. The idea is that they both run `f` as often as possible until it "fails", collecting the results in a list. The difference is that `some f` fails if `f` fails immediately, while `many f` will succeed and "return" the empty list. But what this all means exactly depends on how `<|>` is defined.

Is it only useful for parsing? Let's see what it does for the instances in base: `Maybe`, `[]` and `STM`.

First `Maybe`. `Nothing` means failure, so `some Nothing` fails as well and evaluates to `Nothing` while `many Nothing` succeeds and evaluates to `Just []`. Both `some (Just ())` and `many (Just ())` never return, because `Just ()` never fails! In a sense they evaluate to `Just (repeat ())`.

For lists, `[]` means failure, so `some []` evaluates to `[]` (no answers) while `many []` evaluates to `[[]]` (there's one answer and it is the empty list). Again `some [()]` and `many [()]` don't return. Expanding the instances, `some [()]` means `fmap (():) (many [()])` and `many [()]` means `some [()] ++ [[]]`, so you could say that `many [()]` is the same as `tails (repeat ())`.

For `STM`, failure means that the transaction has to be retried. So `some retry` will retry itself, while `many retry` will simply return the empty list. `some f` and `many f` will run `f` repeatedly until it retries. I'm not sure if this is useful thing, but I'm guessing it isn't.

So, for `Maybe`, `[]` and `STM` `many` and `some` don't seem to be that useful. It is only useful if the applicative has some kind of state that makes failure increasingly likely when running the same thing over and over. For parsers this is the input which is shrinking with every successful match.

edited Oct 6 '11 at 22:47 answered Oct 6 '11 at 22:39
Sjoerd Visscher
9,524 ● 1 ● 32 ● 55

E.g. for parsing (see the "Applicative parsing by example" section).

answered Oct 6 '11 at 7:12
Alexey Romanov
57.5k ● 13 ● 136 ● 268

1 I am not familiar with Parsec. I'd appreciate some explanation. – missingfaktor Oct 6 '11 at 7:41

2 As far as I understand, if you have a parser `p` for X, then `some p` is a parser for 0 or more X and `many p` is a parser for 1 or more X. – Ingo Oct 6 '11 at 9:08

2 @missingfaktor `some` and `many` are implementaed in terms of `<|>`. This combinator is useful also in other ways. Consider `Either` : `Just 0 <|> Just 1 = Just 0`, `Nothing <|> Just 2 = Just 2`, `Just 3 <|> Nothing = Just 3`, `Nothing <|> Nothing = Nothing` – FUZxxl Oct 6 '11 at 10:17

1 @missingfaktor: that is the usual application; I'm not sure if `Alternative` is used for anything else. You could say that "in general", `some` is used whenever you want something to run multiple times (but doesn't **have** to run), and `many` to run at least once. – ivanm Oct 6 '11 at 11:24

2 @Ingo @ivanm Note that you have `some` and `many` backwards. `some` is one or more (i.e. `+` in regexps) and `many` is zero or more (i.e. `*`). – Sjoerd Visscher Oct 6 '11 at 20:24