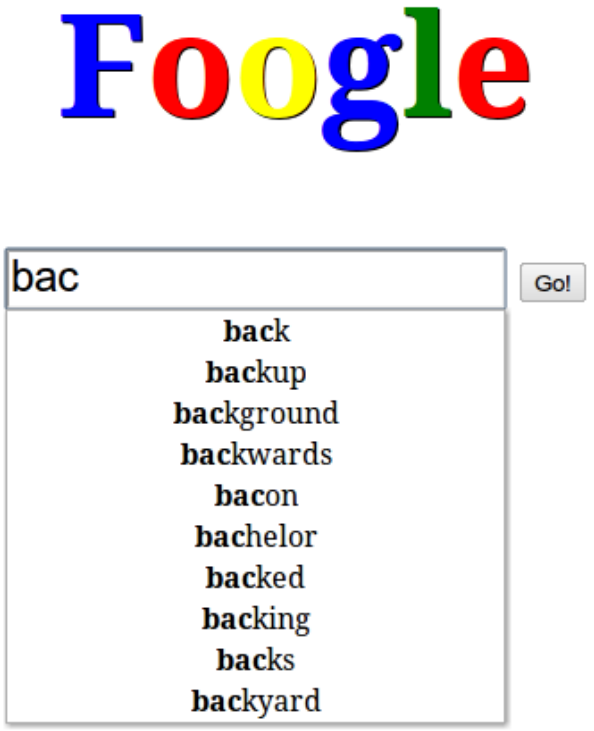
When Haskell is the fastest choice

(if you want to stay sane)

HESP (Haskell Extrasensory Perception)

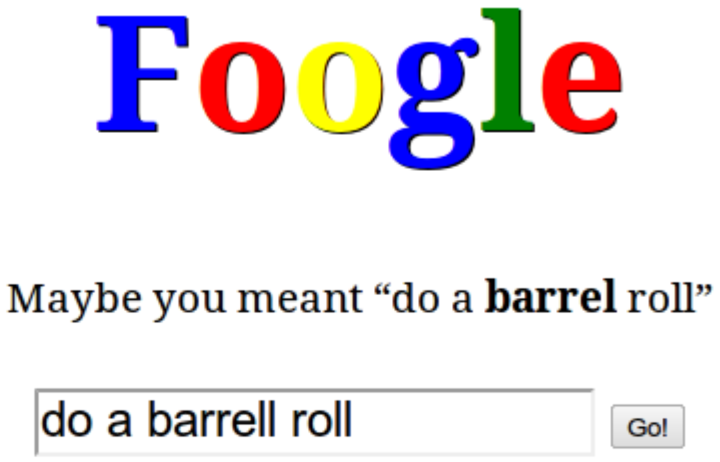
- Basic completion / spell checking server
- Written in Haskell using
  - The Warp web server
  - A neat data structure
  - The YUI JS library for the minimal frontend
- Development time: ~ 10hrs
- ~ 300 loc
- Req/sec: ~20k sec (on my laptop)


Foogle

bac

Go!

back
backup
background
backwards
bacon
bachelor
backed
backing
backs
backyard


Foogle

Maybe you meant “do a **barrel** roll”

do a barrell roll

Go!

The tools

Warp

- High performance Haskell web server
- Based mostly on two concepts:
  - Enumerator pattern
  - Builders
- And GHC 7 new event manager

A word about GHC (Glasgow Haskell Compiler)

- GHC has its own runtime, which provides lightweight threads - similar to Erlang
- As in Erlang, this enables applications that spawn a great number of threads, since the RTS will take care of them
- If this wasn't enough, GHC 7 comes with a new event manager that uses epoll/kqueue and better data structures
- All this functionality is exposed through a high level API
- => Writing scalable applications is easy

The Enumerator pattern

- Fact: lazy IO doesn't scale - you have little control on when the resources are released
- The Enumerator pattern solves this problem
- Iteratees consume data in chunks, signaling if they finish or if they need more data.
- Enumerators feed data into Iteratees
- The logic is captured in the Step data type

Step

Omitting the monadic part:

```
data Stream data =
    Chunks [data]  Chunks of data
  | EOF  End of file

data Step data result =
                                    Need more data to continue
    Continue (Stream a -> Step data result)
  | Yield result (Stream a)  Finished computing, returning leftover data
  | Error SomeException  Some error occured (we got an EOF too soon, etc)
```

Why all this?

- Warp IO model is based on Enumerators
- A Warp application is essentially something of the type `Request -> Step ByteString Response`, where the ByteString chunks contain the body of the request and the Request data type contains the headers
- When a new connection is estabilished, an handler thread is spawn and fed 4KB chunks until it yields something
- The leftover is fed to the next action

What about the response?

Builders

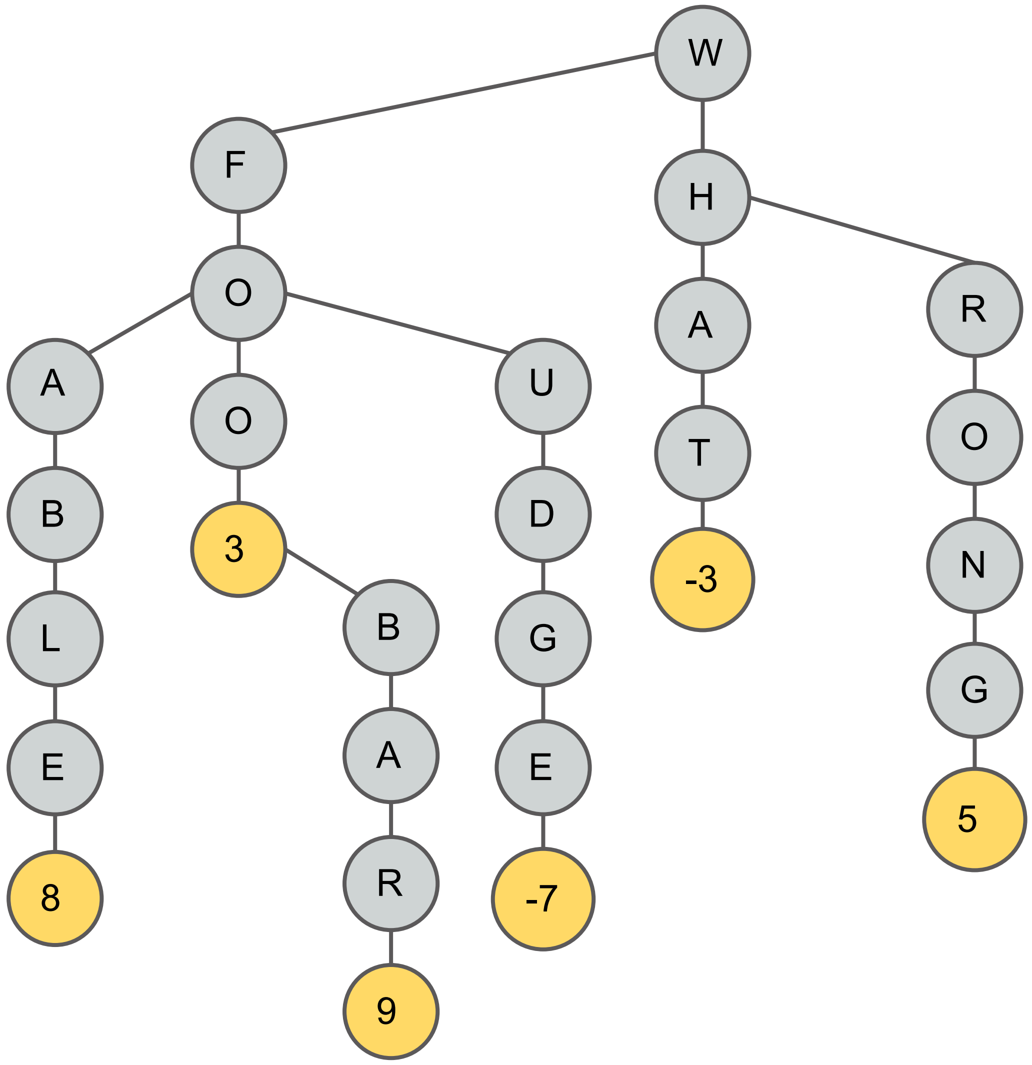
- Builders are an efficient way to build byte streams, based on difference lists
- A difference list is a function that given a list, returns the original contents of the difference list prepended at the given list.
- This enables O(1) appending, doing the "flattening" once - just when we need it
- Chunk the data smartly when flattening

Builders

- All Warp responses are returned in Builder form, for two reasons:
  - The HTTP response headers can be efficiently prepended to the response body
  - The data is packed efficiently in 32KB chunks (in the form of a lazy ByteString) so for most requests we need just one system call

Suggestion and spell checking

Ternary Search Trie



what, -3
foo, 3
foobar, 9
fable, 8
wrong, 5
fudge, -7

Ternary Search Trie

- One of the best pure data structure to store dictionaries of strings (or any list-like element)
- Extremely simple and generic implementation (around 80 lines)
- Fast lookup - O(m + log(n)) where m is the length of the string
- And most importantly we can implement two very useful operations for what we're doing

Ternary Search Trie

- The "prefix" operation, gives all the strings that contain the given string as a prefix
- The "match" operation matches a wildcarded string, for example `match "*l*e" dict` might return "aloe", "blue", and "glue" - with the respective values.

The dictionary

- The dictionary used is a frequency ordered dictionary list extracted from TV and movie scripts (thanks wiktionary)
- We load it assigning to each word its position in the list, so we have a mapping String -> Int
- Lower value means higher frequency

Suggestions

- When the user starts typing, we look up all the words that have the given prefix, sort them and return the 10 most frequent
- These results are cached in another mapping of type String -> [String]
- The cache is shared amongst all the request in a IO reference that can be modified atomically

Spell checking

- The spell checking is trickier, and could be improved
- After a user has submitted the form, we check if the word is in the dictionary
- If it isn't we generate various permutations of the input:
    - Delete each of the letter
    - Swap adjacent letters
    - Replace each letter with something else (wildcard it)
    - Insert letters (wildcards)

Spell checking

- For example, for the word "word" we have the following permutations:

"ord", "wrd", "wod", "wor", "owrd", "wrod", "wodr", "*ord", "w*rd", "wo*d", "wor*", "*word", "w*ord", "wo*rd", "wor*d"

- The fact that we can look up wildcarded words makes this approach drammatically faster - we would have to manually insert letters and look them up otherwise

Spell checking

- If none of the permutations are present, we generate the permutations of the permutations, and look those up
- If we still don't have a result, we give up
- If we have candidates, we pick the one with the highest frequency
- Again, the results are cached
- Both the algorithms should be improved by adjusting the frequency as the users search for words

The interface

Server-client communication

- The server accepts queries in two locations (/suggest.json and /correct.json).
- Unsurprisingly, it returns JSON objects, that are then parsed by the JavaScript frontend
- suggest.json accepts a word to complete and returns an array with a maximum of 10 words
- correct.json accepts a phrase and returns the same phrase correcting the words that it could correct

Server-client communication

```
$ curl http://localhost:3000/suggest.json?q=wat
["watch","water","watching","watched","watches","waters","
water's","watcher","watson","watchin"]

$ curl http://localhost:3000/correct.json?q=gerat%20sucess
great success
```

The frontend

- Minimal frontend included to demonstrate the server.
- Using the excellent YUI JavaScript interface library, which includes
  - An AutoComplete widget for form inputs
  - A nice networking interface to do remote requests
  - A JSON parser
  - ... a lot more

Demo, Q&A