Functional Reactive Programming

Brandon Siegel
Senior Engineer, Mobile Defense

Topics

- Monads
- Futures and Promises
- Observables, Observers, Subscriptions, and Subjects
- Functional Reactive Programming in

What is a monad?

What is a monad?

“A monad is just a monoid in the category of endofunctors.” – Some Jackass

What is a monad?

"A monad is just a monoid in the category of endofunctors." – Some Jackass

- Monoid?
- Category theory??
- Functors!?
- Sounds hard, let's go shopping

Let’s try again

What is a monad?

“The key to understanding monads is that they are like burritos.” – Brent Yorgey

“Monads are space suits that let us safely travel from one function to another.” – Eric Kow

What is a monad?

“The key to understanding monads is that they are like burritos.” – Brent Yorgey

“Monads are space suits that let us safely travel from one function to another.” – Eric Kow

- Nope.

Let’s take a step back

Monads are a tool used in functional programming, so why are they used there? What makes functional programming so… functional?

Let’s take a step back

Monads are a tool used in functional programming, so why are they used there? What makes functional programming so… functional?

Functional programming avoids dangerous global mutable state by making functions and function composition first class features.

Functions

If we can limit any side-effects to the scope of a single function, we can be certain that calling that function is idempotent.

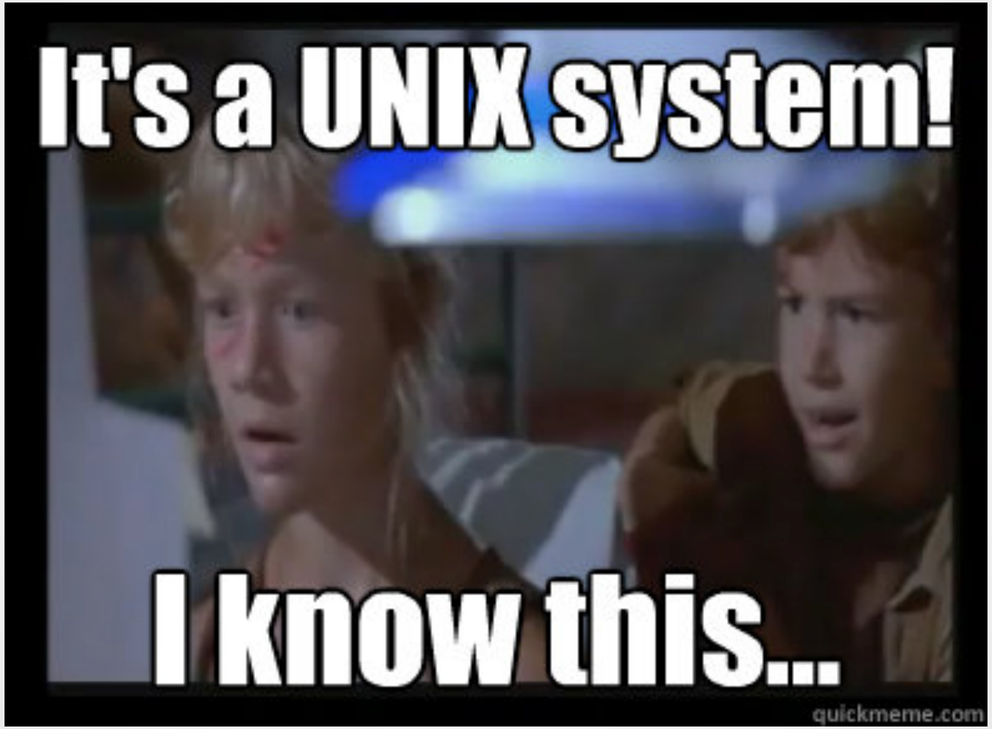
Calling *f(x)* many times produces the same result each time, and doesn’t affect the scope of the caller.

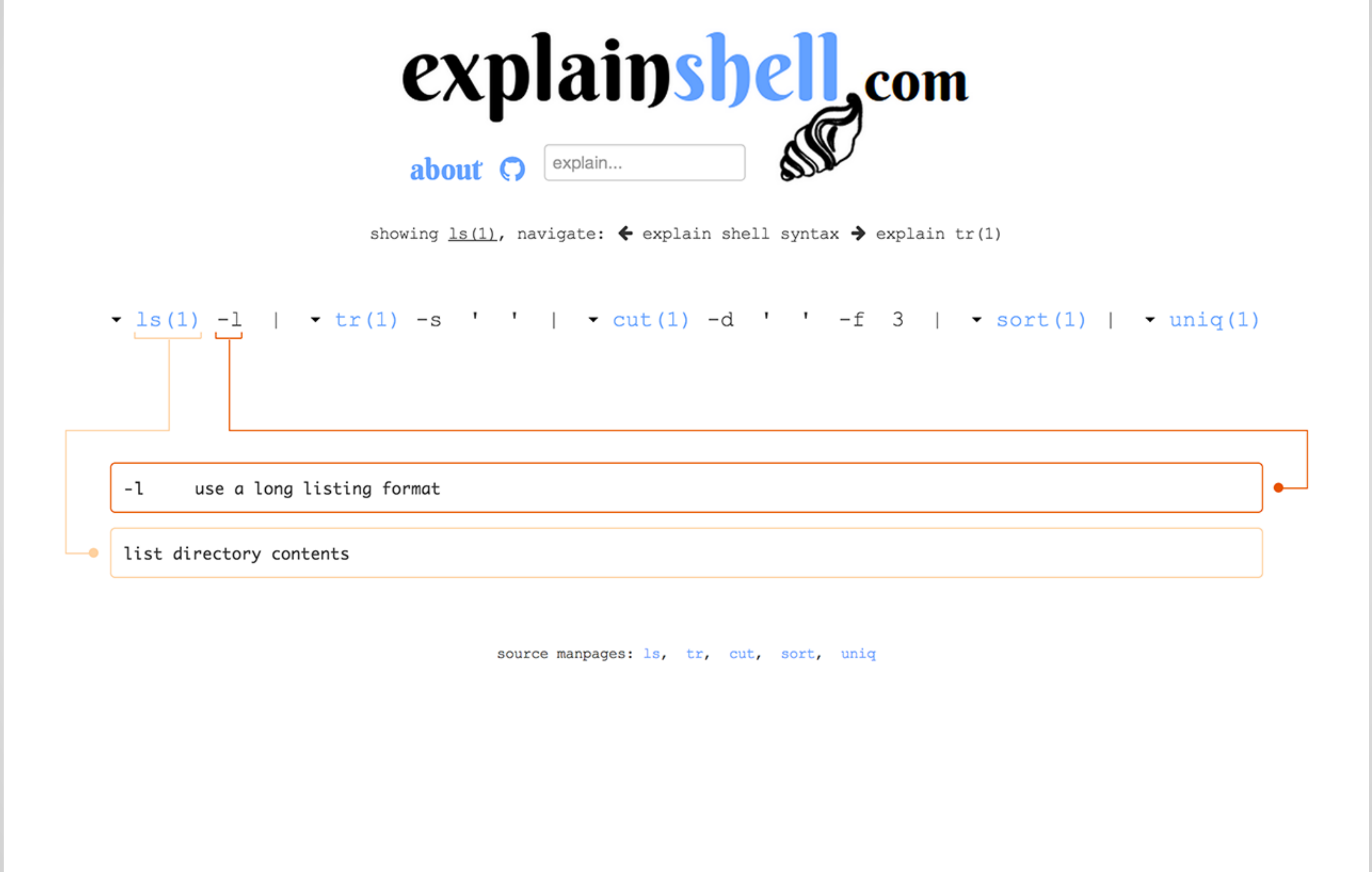
Function composition

If we know that calling functions is idempotent, we can chain multiple functions together to produce complex behavior that is side-effect free.

Wait, simple idempotent functions that do one thing, chained together to perform complex operations…?

```
ls -l | tr -s ' ' | cut -d ' ' -f 3 | sort | uniq
```





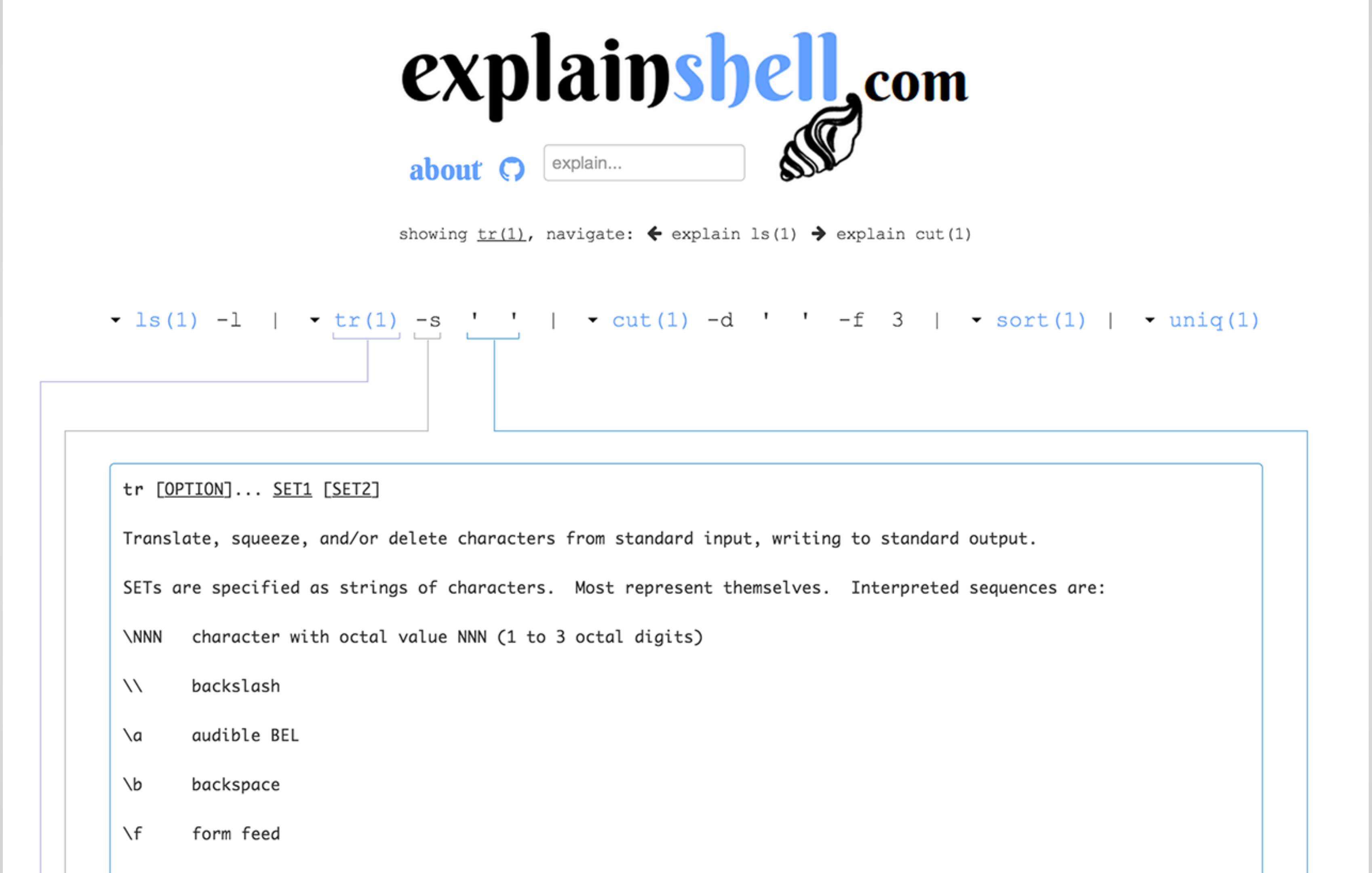
showing ls(1), navigate: ← explain shell syntax → explain tr(1)

```
ls(1) -l | tr(1) -s ' ' | cut(1) -d ' ' -f 3 | sort(1) | uniq(1)
```

```
-l      use a long listing format
```

```
list directory contents
```

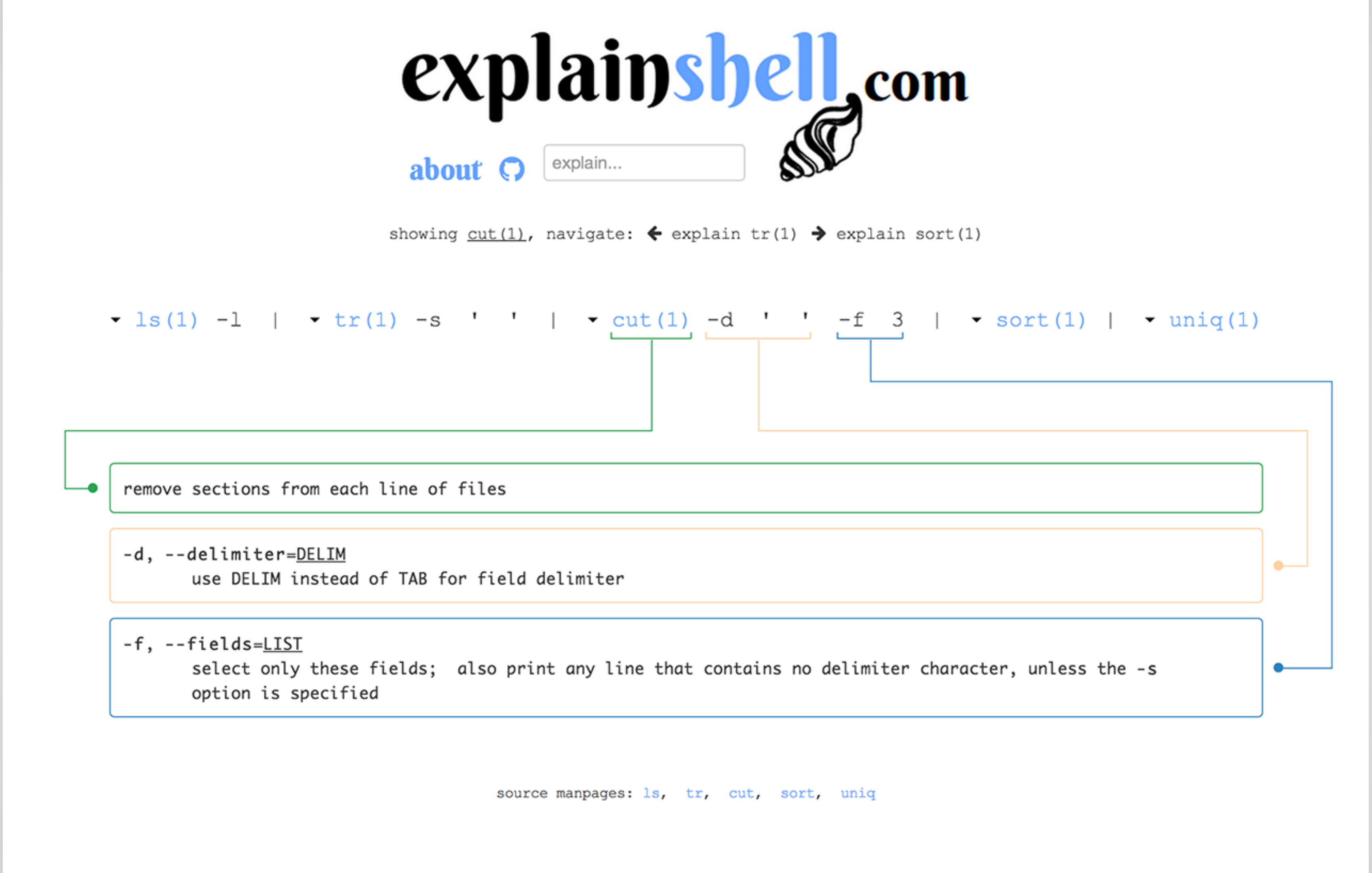
source manpages: ls, tr, cut, sort, uniq



showing tr(1), navigate: ← explain ls(1) → explain cut(1)

```
ls(1) -l | tr(1) -s ' ' | cut(1) -d ' ' -f 3 | sort(1) | uniq(1)
```

```
tr [OPTION]... SET1 [SET2]

Translate, squeeze, and/or delete characters from standard input, writing to standard output.

SETs are specified as strings of characters.  Most represent themselves.  Interpreted sequences are:

\NNN   character with octal value NNN (1 to 3 octal digits)

\\     backslash

\a     audible BEL

\b     backspace

\f     form feed
```



showing cut(1), navigate: ← explain tr(1) → explain sort(1)

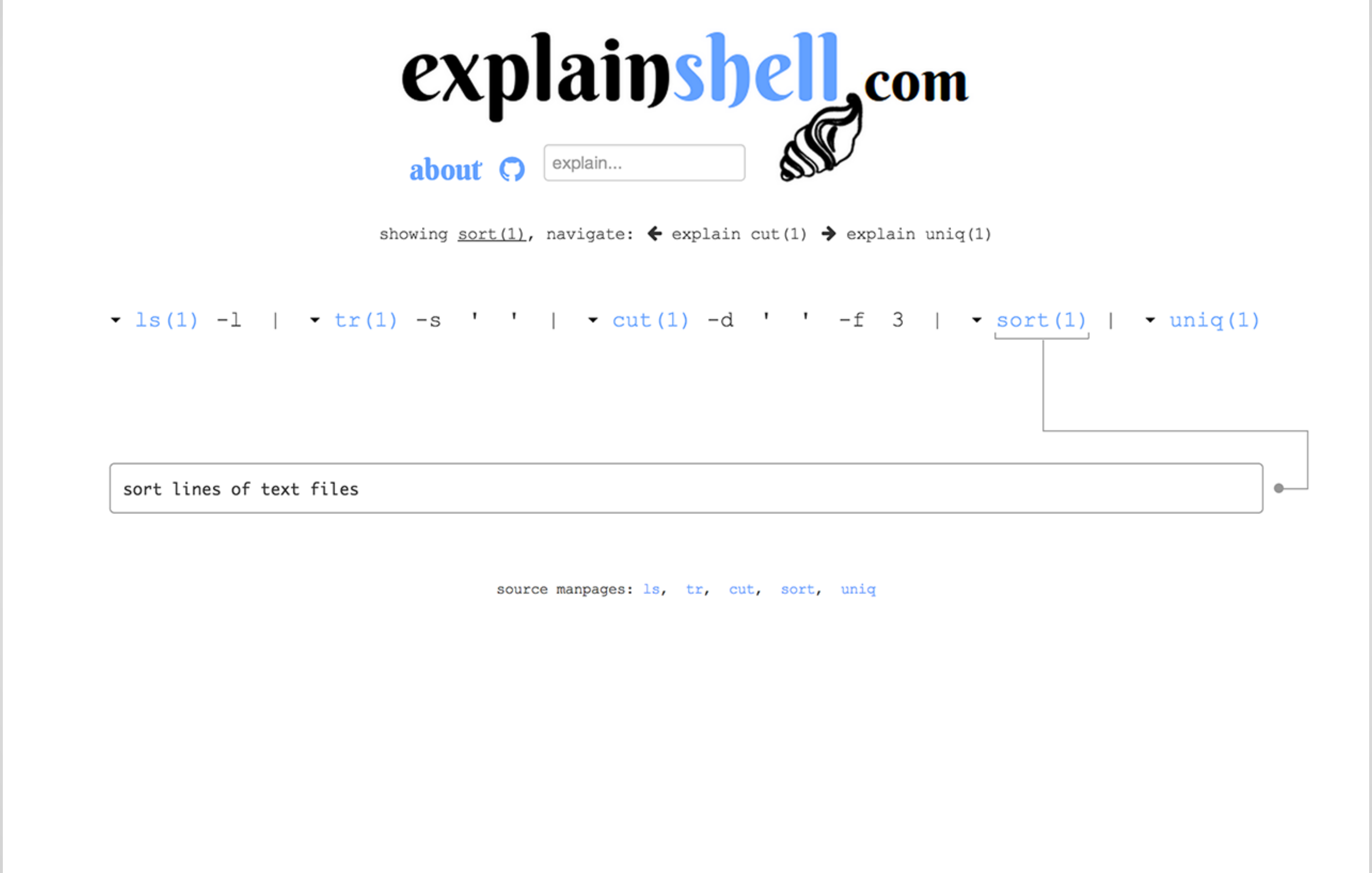
```
ls(1) -l | tr(1) -s ' ' | cut(1) -d ' ' -f 3 | sort(1) | uniq(1)
```

remove sections from each line of files

```
-d, --delimiter=DELIM
       use DELIM instead of TAB for field delimiter
```

```
-f, --fields=LIST
       select only these fields;  also print any line that contains no delimiter character, unless the -s
       option is specified
```

source manpages: ls, tr, cut, sort, uniq



explainshell.com

about

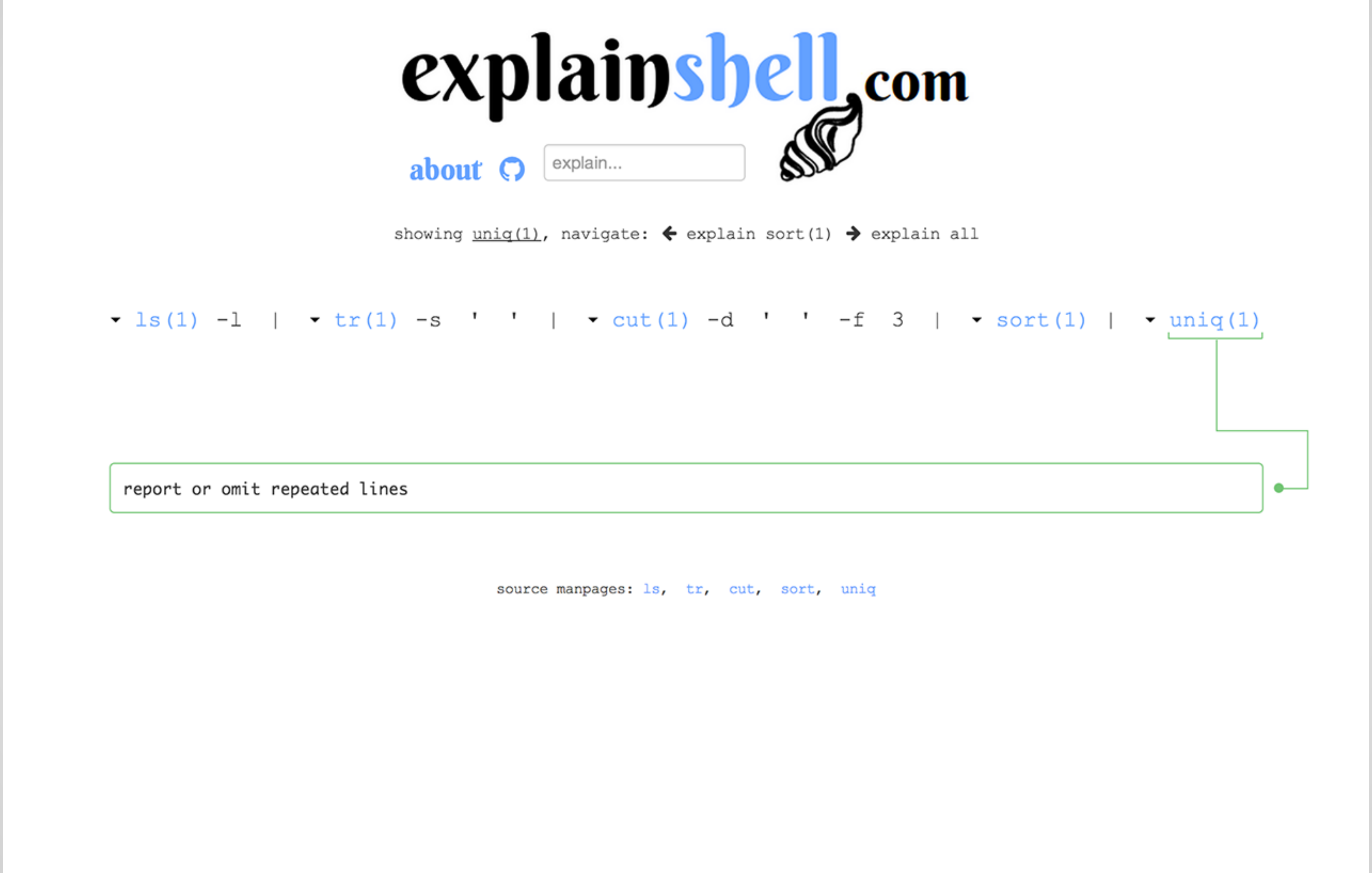
explain...

showing sort(1), navigate: ← explain cut(1) → explain uniq(1)

```
ls(1) -l | tr(1) -s ' ' | cut(1) -d ' ' -f 3 | sort(1) | uniq(1)
```

sort lines of text files

source manpages: ls, tr, cut, sort, uniq

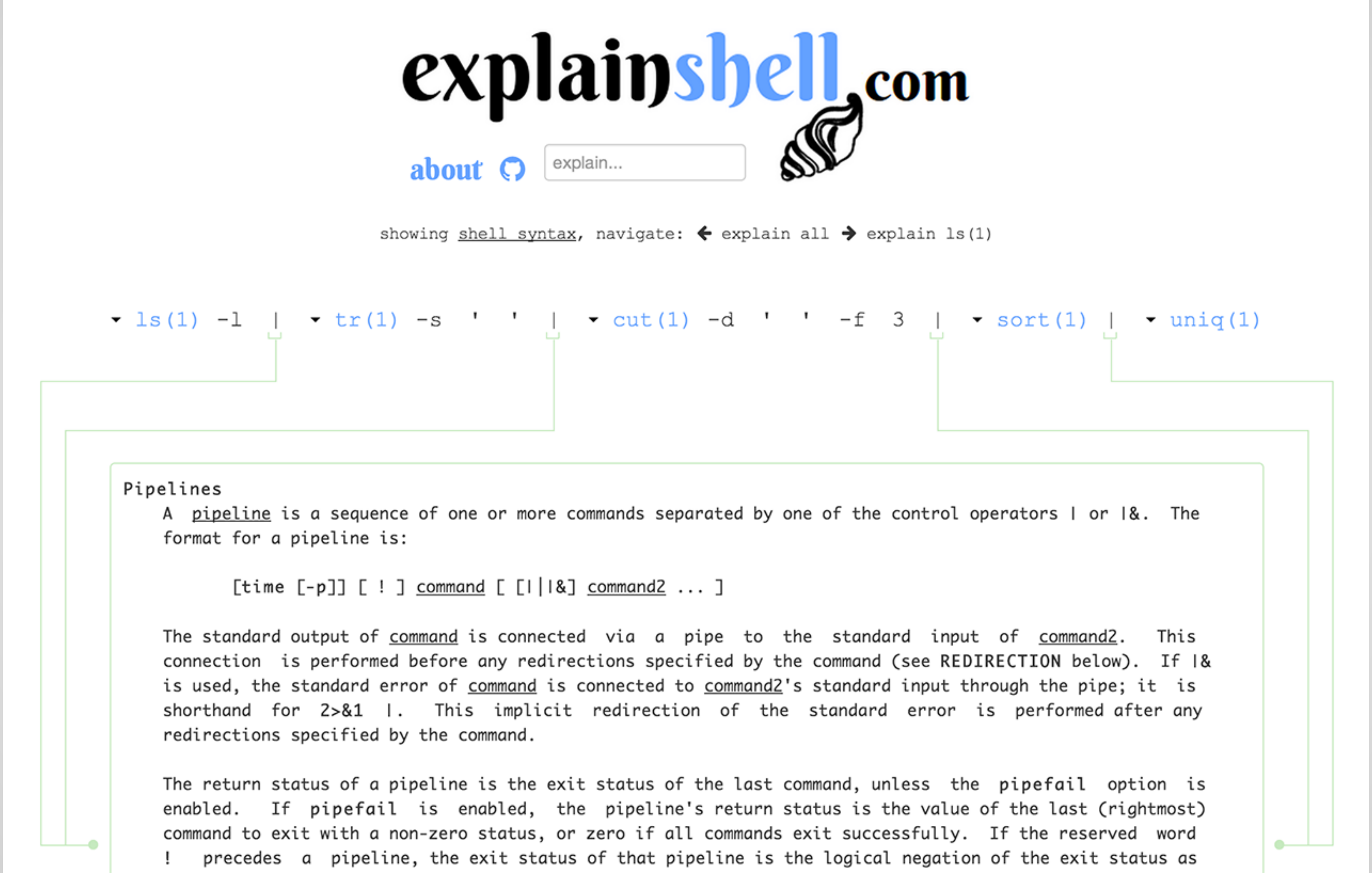


showing uniq(1), navigate: ← explain sort(1) → explain all

```
ls(1) -l | tr(1) -s ' ' | cut(1) -d ' ' -f 3 | sort(1) | uniq(1)
```

report or omit repeated lines

source manpages: ls, tr, cut, sort, uniq



showing shell syntax, navigate: ← explain all → explain ls(1)

```
ls(1) -l | tr(1) -s ' ' | cut(1) -d ' ' -f 3 | sort(1) | uniq(1)
```

Pipelines

A pipeline is a sequence of one or more commands separated by one of the control operators | or |&. The format for a pipeline is:

```
[time [-p]] [ ! ] command [ [|||&] command2 ... ]
```

The standard output of command is connected via a pipe to the standard input of command2. This connection is performed before any redirections specified by the command (see REDIRECTION below). If |& is used, the standard error of command is connected to command2's standard input through the pipe; it is shorthand for 2>&1 |. This implicit redirection of the standard error is performed after any redirections specified by the command.

The return status of a pipeline is the exit status of the last command, unless the pipefail option is enabled. If pipefail is enabled, the pipeline's return status is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands exit successfully. If the reserved word ! precedes a pipeline, the exit status of that pipeline is the logical negation of the exit status as

What is a monad?

What is a monad?

Monads are tools that allows you to:

- take some **data**
- apply a series of **transformations**

and do so in a way that is idempotent and that encapsulates side effects.

Data

To understand what the result of a computation will be, we must consider not only the data we are operating on but we must also anticipate the effects that carrying out the computation may cause.

Effects of computations

- A computation may not produce a result
- A computation may fail
- A computation may operate on and produce multiple values
- A computation may take some time to run
- A computation may produce multiple values after varying amounts of time

How do we handle effects?

- A computation may not produce a result

```
if (foo != null) {
  ...
}
```

How do we handle effects?

- A computation may fail

```
try {
   ...
} catch (Exception e) {
   ...
}
```

How do we handle effects?

- A computation may operate on or produce multiple values

```
string[] strs = ...;
for (int i=0; i<strs.length; i++) {
   ...
}
```

How do we handle effects?

- A computation may take some time to run

```
HttpClient c = new HttpClient(...);
c.get(url, new ResponseHandler() {
   @override public void onSuccess(...) {
      ...
   }
});
```

How do we handle effects?

- A computation may produce multiple values after varying amounts of time

```
public event
EventHandler Clicked;

...

if (Clicked != null) {

    Clicked(this,

    new EventArgs())

}
```

```
public void Button_Clicked(
object sender, EventArgs e) {

    ...

}

...

widget.Clicked += new
EventHandler(Button_Clicked);
```

How do we handle effects?

As developers, we anticipate the effects of our computations and use well-known patterns to deal with them. However, it’s all too easy to forget about a potential effect. Also, without digging into our source code, callers have no way to know what effects calling our functions may cause.

Explicitly calling out effects

If we encode the potential effects into the type of the result, it solves two problems at once:

- The potential effects are part of the public contract – callers know what effects may occur when calling a function
- The compiler can check the type and warn us if we've forgotten to handle a potential effect

Data (+ effects) = type signature

A computation may not produce a result	`Option[T]`
A computation may fail	`Try[T]`
A computation may operate on and produce multiple values	`Enumerable[T]`
A computation may take some time to run	`Future[T]`
A computation may produce multiple values after varying amounts of time	`Observable[T]`

What is a monad?

Monads are tools that allows you to:

- take some **data**
- apply a series of **transformations**

and do so in a way that is idempotent and that encapsulates side effects.

Transformations = functions

If we have a monad object, we can apply transformation functions to it that will transform the value(s) but preserve the semantics of the monad.

Useful but unexpected bonus: the function(s) will not be applied until we ask for a result!

Transformations = functions

- `filter`
- `map`
- `flatMap`
- `reduce`
- `foldLeft / foldRight`

Composing functions on Enumerable

```
val nums = (1 to 100)
nums.filter(n => n%2 == 0)
     .map(n => n*n)
     .take(5)
     .foreach(n => print(n + “ ”))

>> 4 16 36 64 100
```

Composing functions on Enumerable

```
val nums = Stream.from(1)
nums.filter(n => n%2 == 0)
    .map(n => n*n)
    .take(5)
    .foreach(n => print(n + “ ”))

>> 4 16 36 64 100
```

Composing functions on Try

```
val divisor = 5
val items = (1 to 10)
val result = Try { 15 / divisor }
result.map(n => n*2)
      .flatMap(n => Try { items(n) })

>> Success(7)
```

Composing functions on Try

```
val divisor = 3
val items = (1 to 10)
val result = Try { 15 / divisor }
result.map(n => n*2)
      .flatMap(n => Try { items(n) })

>> Failure(IndexOutOfBoundsException)
```

Composing functions on Try

```
val divisor = 0
val items = (1 to 10)
val result = Try { 15 / divisor }
result.map(n => n*2)
      .flatMap(n => Try { items(n) })

>> Failure(ArithmeticException)
```

So...

Monads let us ignore certain effects of computations and just focus on the core of what we want to get done. They allow us to apply a sequence of computations to a value, call out potential side effects in their type signature, and provide a representation of both the result of the computations and the actual side effects that occurred as their result.

Category theory! bind()! apply()!

There are subtleties to what technically is or is not a monad, what properties they must exhibit, and so on that are based in the mathematics of category theory. Knowing this is not useful to learning what monads are and how to use them, so don’t get tripped up in the textbook definition of a monad – if it acts like our description of a monad, it’s close enough!

Erik Meijer

- Pioneered and popularized FRP
- Created LINQ and Reactive Extensions
- Influenced design of F#, C#, Haskell

Monadic Types

	Single Result	Multiple Results
Synchronous	`Try[T]`	`Enumerable[T]`
Asynchronous	`Future[T]`	`Observable[T]`

Monadic Types

	Single Result	Multiple Results
Synchronous	`Try[T]`	`Enumerable[T]`
Asynchronous	**`Future[T]`**	**`Observable[T]`**

Future handles failures and latency

A Future allows us to attach code to an asynchronous computation that will run when that computation is complete. Used naively you can still end up with ‘callback hell’ but because Future is a monad, you can apply combinator functions to schedule additional Futures and transform intermediate results to obtain a final result.

Future

```
val result = Future {
  apiClient.getPosts(user)
}
result.onSuccess { posts => ...}
result.onFailure { e => ... }
```

Future

```
val result = Future {
  client.getPosts(user)
}.flatMap { posts =>
  val post = highestRated(posts)
  Future { apiClient.getComments(post) }
}
result.onSuccess { comments => ... }
```

Operations on Future

- onComplete / onSuccess / onFailure
- recover / recoverWith / fallbackTo

- Await.ready / Await.result

Promises

A Future object allows a consumer to operate on the result of an asynchronous computation.

The producer side of the producer / consumer relationship is implemented using a Promise.

Promises

Promises are synchronization points between producers and consumers. Producers can insert a value, and consumers can acquire a future from the promise that completes when a value is assigned. A promise can only ever be completed once, so completing a promise is thread-safe.

Promises

```
val p = Promise[List[Post]]
val result = p.future
...
result.onComplete { ... }
...
p.success(apiClient.getPosts(user))
```

Let’s talk about

Functional Reactive Programming

Reactive

Reactive programming is a style of programming in which **data** controls the execution of the program. Contrast this with typical imperative programming where **statements** (such as `if` and `for`) control the flow of execution.

Functional

Functional in this case simply means that we’ll use functional concepts (composing transformations on monads) instead of procedural concepts like callbacks or events to implement a reactive programming model.

Functional Reactive Programming

The decision to use functional vs. procedural, or reactive vs. imperative programming techniques are two separate arguments. However, both functional and reactive programming styles seem to provide benefits over their alternatives, and it turns out they actually compliment each other quite well.

Observable

Observables are the foundation of functional reactive programming. They are collections that asynchronously produce their values. Like Enumerable, Observable produces a series of values. Like Future, any of these values may be produced at some arbitrary point in the future.

Static Computation

```
val b = 1
val c = 2
val a = b + c
```

Reactive Computation

```
val bs = Observable(...)
val cs = Observable(...)
val as = bs.combineLatest(cs,
              (b, c) => b + c
         )

as.subscribe(a => ...)
```

Observable, Observer, Subscription

- Observable exposes a stream of events which Observers can subscribe to
- When an Observer subscribes to an Observable, it obtains a Subscription
- An Observer can unsubscribe from a Subscription when it is no longer interested in receiving results

Observable

```
val trades: Observable[Trade] = ...
val suspectTrades =
  trades.buffer(2, 1)
        .filter(pair => isSuspect(pair))
        .map(
          pair => (pair(0).id,pair(1).id)
        )
```

Subscriptions

```
val subscription = suspectTrades.
subscribe(suspects => ... )
...
subscription.unsubscribe()
```

Observer

For Enumerable, we needed one handler (`forEach`) to handle the results because we only had one situation we had to react to (the next element has been produced, aka `onNext`). For Future, we had two situations (success and failure) and so we needed two handlers, `onSuccess` and `onFailure`.

Observer

With Observable, we now have three states. This starts to get unwieldy, so for convenience we bundle them up into an Observer. Observer objects define three functions: `onNext`, `onError`, and `onCompleted`. When an Observer subscribes to an Observable, that Observable informs the Observer about its activity by calling these functions according to the following contract.

The Rx contract

- To produce value, the Observable may call `onNext` zero or more times
- To signal an error, the Observable may call `onError` at most once
- To signal completion, the Observable may call `onCompleted` at most once
- No calls to `onNext` may be made after the Observable calls `onError` or `onCompleted`

Subject

Just like a Promise acts as a synchronization point between a producer and consumer of a Future, a Subject acts as a synchronization point between an Observable that is producing values and Observers that consume them. It acts like a fan-out channel to forward results from a single Observable to many Observers.

Subject

Subjects implement both the Observable and Observer interfaces. You can put results into a Subject by calling the `onNext` / `onError` / `onCompleted` functions of its Observer interface, and a Subject can be subscribed to by calling the `subscribe` function of its Observable interface.

Types of subjects

There are many types of Subjects which provide different semantics for deciding which events are sent or replayed to subscribers. For example, they may buffer some or all events they have received so that Observers that subscribe later can receive a replay of these events.

Operations on Observable

- `concat / merge`
- `buffer / groupBy`
- `onErrorResumeNext / onErrorReturn`
- `retry / timeout`

- `toBlockingObservable`

https://github.com/Netflix/RxJava/wiki/Observable

Reactive Programming in Practice

MVVM, Rx, and ReactiveUI in the Windows Phone client

MVVM Interfaces

```
interface INotifyPropertyChanged {
  event PropertyChangedEventHandler PropertyChanged;
}
```

```
public string Id {
  get { return _id; }
  set {
    _id = value;
    Changed(“Id”);
  }
}
```

```
public void Changed(string propName) {
  if (PropertyChanged != null) {
    PropertyChanged(this,
      new PropertyChangedEventArgs(
        propName));
  }
}
```

MVVM Interfaces

```
interface ICommand {
  bool CanExecute();
  void Execute();
  event EventHandler CanExecuteChanged;
}
```

Rx – Reactive Extensions

- First common* implementation of FRP
- Influenced creation of RxJava / ReactiveCocoa
- `subscribeOn(Thread)` / `observeOnDispatcher` ensures producer (observable) code runs in background thread but consumer (observer) code runs on the UI thread

ReactiveUI

- C# MVVM framework based on Observables instead of events
- Allows complex cross-event behavior to drive UI changes and Commands’ ability to execute
- ReactiveCollections hold ReactiveObjects and notify when items are added, removed, changed or emit events
- Works on Android / iOS / Mac with Xamarin

WP – composing UI observables

```
_loginCommand = new ReactiveCommand(this.WhenAny(
                i => i.Busy,
                i => i.Email,
                i => i.Password,
                i => i.PhoneNumber,
                (b, e, p, n) =>
                  !b.Value &&
                  !String.IsNullOrEmpty(e.Value) && !String.
                  IsNullOrEmpty(p.Value) && !String.
                  IsNullOrEmpty(n.Value) && ValidateAll()));
_loginCommand.Subscribe(DoLoginAction);
```

WP – creating derived UI collections

```
public ReactiveCollection<MenuItem> EnabledItems {
  get { return _enabledItems; }
}

...

_menuItems.ChangeTrackingEnabled = true;

_enabledItems = _menuItems.CreateDerivedCollection(i => i,
                                                   i => i.
IsEnabled);
```